CENTRO UNIVERSITÁRIO UNIVATES
CURSO DE ENGENHARIA DA COMPUTAÇÃO

ESTUDO COMPARATIVO ENTRE OS SISTEMAS GERENCIADORES DE BANCO DE DADOS POSTGRESQL E MONGODB PARA O ARMAZENAMENTO E BUSCA DE METADADOS MARC

Jader Osvino Fiegenbaum

Lajeado, junho de 2016

Jader Osvino Fiegenbaum

ESTUDO COMPARATIVO ENTRE OS SISTEMAS GERENCIADORES DE BANCO DE DADOS POSTGRESQL E MONGODB PARA O ARMAZENAMENTO E BUSCA DE METADADOS MARC

Trabalho de Conclusão de Curso II, apresentado ao curso de Engenharia da Computação, do Centro Universitário UNIVATES, para obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Prof. Me. Evandro Franzen

Lajeado, junho de 2016

RESUMO

A utilização de sistemas de gestão de acervo reduz o trabalho e o custo interno das bibliotecas, além de recuperar mais rapidamente a informação para os usuários. Atualmente, a utilização desses sistemas é fundamental, principalmente devido ao volume de informação, a qual deverá ser catalogada e recuperada. O surgimento do formato MARC em 1960 é um dos fatores que possibilitou o desenvolvimento desses sistemas, pois é um marco que representa a transição da catalogação em fichas catalográficas para computador. Apesar do MARC apresentar uma estrutura flexível e dinâmica, muitos sistemas de gestão de acervos utilizam bancos de dados relacionais, que em sua maioria utilizam estruturas mais rígidas para o armazenamento de dados. O propósito deste estudo é desenvolver um protótipo para comparar quantitativamente e qualitativamente a adesão e performance dos SGDB's PostgreSQL e MongoDB para o armazenamento e busca de metadados MARC. Neste estudo serão abordados conceitos de dados e metadados, bem como sua visão geral, os principais conceitos dos modelos NoSQL e relacional e as principais características dos SGDB's PostgreSQL e MongoDB.

Palavras-chave: PostgreSQL. MongoDB. Metadados. Marc. Protótipo.

ABSTRACT

The use of archive management systems reduces the work and cost of internal libraries, and recover faster information for users. At the present time, the use of such systems is critical, especially due to the amount of information which must be cataloged and retrieved. The emergence of the MARC format in 1960 is one of the factors that enabled the development of these systems, it is a landmark that represents the transition from cataloging catalog cards to computer. Despite the MARC present a flexible and dynamic structure, many collections management systems using relational databases, which mostly use more rigid structures for data storage. The purpose of this study is to develop a prototype to compare quantitatively and qualitatively the adhesion and performance of DBMS's PostgreSQL and MongoDB for storage and search MARC metadata. This study will address concepts of data and metadata, as well as its overview, the main concepts of NoSQL and relational models and the main features of the DBMS's PostgreSQL and MongoDB.

keywords: PostgreSQL. MongoDB. Metadata. Marc. Prototype.

LISTA DE FIGURAS

Figura 1 – Linguagem de marcação.....18
Figura 2 – Campos básicos do formato MARC 21.....20
Figura 3 – Registro catalográfico do livro “A assustadora história da maldade”.....20
Figura 4 – Relação Alunos.....22
Figura 5 – *Map*/*Reduce* na coleção sites.....29
Figura 6 – Resultado do *MapReduce* na coleção sites.....29
Figura 7 – Registro de uma tabela banco de dados relacional armazenada em disco.....31
Figura 8 – Registro de uma tabela banco de dados orientado a colunas.....31
Figura 9 – Documento JSON contendo dados da coleção de periódicos Super Interessante…33
Figura 10 – Registro de chave/valor.....34
Figura 11 – Estrutura de um objeto JSON.....43
Figura 12 – Estrutura de um vetor em JSON.....44
Figura 13 – Equivalência entre SGDB relacional e MongoDB.....48
Figura 14 – Diagrama ER para PostgreSQL.....54
Figura 15 – Coleção de documentos onde estão armazenados os metadados do objeto/material 54
Figura 16 – Coleção de documentos onde serão são armazenados os metadados disponíveis.55
Figura 17 – Coleção de documentos onde são armazenados os tipos de materiais.....56
Figura 18 – Coleção de documentos onde são armazenados os tipos físico de materiais.....56
Figura 19 – Coleção de documentos onde são armazenados os resultados dos testes.....56
Figura 20 – Interface para cadastro de tipos.....57
Figura 21 – Interface para cadastro de materiais.....58
Figura 22 – Interface de pesquisa de materiais.....58
Figura 23 – Novo teste completo de desempenho.....59
Figura 24 – Interface do teste de tolerância a falhas.....59
Figura 25 – Interface de visualização de resultados de testes.....60
Figura 26 – Configurações utilizadas no PostgreSQL.....63
Figura 27 – Busca de materiais no PostgreSQL.....64
Figura 28 – Busca de materiais no MongoDB.....65
Figura 29 – Busca de referências para tipo e tipo físico.....65
Figura 30 – Exemplo de contagem de registros no PostgreSQL.....66

Figura 31 – Exemplo de contagem de registros no MongoDB.....67
Figura 32 – Criação do índice test_index para PostgreSQL.....67
Figura 33 – Criação do índice para MongoDB.....67
Figura 34 – Algoritmo de carga 1 para PostgreSQL.....69
Figura 35 – Algoritmo de carga 1 para MongoDB.....69
Figura 36 – Algoritmo de carga 2 para PostgreSQL.....70
Figura 37 – Algoritmo de carga 2 para MongoDB.....71
Figura 38 – Gráfico de resultados do teste de carga 1.....72
Figura 39 – Utilização de espaço em disco do teste de carga 1.....73
Figura 40 – Gráfico de resultados do teste de carga 2.....74
Figura 41 – Utilização de espaço em disco do teste de carga 2.....74
Figura 42 – Algoritmo de volume para PostgreSQL.....75
Figura 43 – Algoritmo de volume para PostgreSQL.....75
Figura 44 – Gráfico do teste de volume.....76
Figura 45 – Algoritmo de volume para PostgreSQL.....77
Figura 46 – Algoritmo de volume para PostgreSQL.....77
Figura 47 – Gráfico do teste de stress.....78
Figura 48 – Algoritmo de tolerância a falhas para PostgresSQL.....78
Figura 49 – Algoritmo de tolerância a falhas para PostgresSQL.....79
Figura 50 – Método para inserir tipo de material no PostgreSQL.....81
Figura 51 – Método para inserir tipo de material no MongoDB.....81

LISTA DE TABELAS

Tabela 1 – Resultado do algoritmo 1 em função do tempo e tamanho.....37
Tabela 2 - Resultado do algoritmo 2 em função do tempo.....37
Tabela 3 – Resultado do algoritmo de teste de volume.....38
Tabela 4 - Resultado do algoritmo de teste de stress.....38
Tabela 5 – Performance do PostgreSQL.....39
Tabela 6 – Performance do MongoDB.....40
Tabela 7 – Limitações do PostgreSQL.....46
Tabela 8 – Resultado do teste de busca.....68
Tabela 9 – Resultado do algoritmo 1 de carga.....72
Tabela 10 – Resultado do algoritmo 2 de carga.....73
Tabela 11 – Resultado do teste de volume.....76
Tabela 12 – Resultado do teste de stress.....77
Tabela 13 – Resultado do teste de tolerância a falhas.....79

LISTA DE ABREVIATURAS

ACID	*Atomicity Consistency Isolation Durability*
API	*Application Programming Interface*
BSD	*Berkeley Software Distribution*
CAP	*Consistency Availability Partition tolerance*
CSV	*Comma-Separated Values*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
ISO	*International Organization for Standardization*
JSON	*Javascript Object Notation*
MARC	*Machine Readable Cataloging*
MVC	*Model View Controller*
MVCC	*Multiversion Concurrency Control*
OLAP	*Online Analytical Processing*
OLTP	*Online Transaction Processing*
RAM	*Random Access Memory*

SGBD	*Sistema de Gerenciamento de Banco de Dados*
SGML	*Standard Generalized Markup Language*
SQL	*Structured Query Language*
SSL	*Secure Sockets Layer*
URL	*Uniform Resource Locator*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO.....11
1.1 Motivação.....13
1.2 Objetivos.....14
1.2.1 Objetivos específicos.....14
1.3 Organização do trabalho.....14

2 REFERENCIAL TEÓRICO.....16
2.1 Dado.....16
2.2 Metadado.....16
2.2.1 MARC.....29
2.3 Sistemas de banco de dados.....22
2.4 Modelo relacional.....23
2.4.1 Transações.....23
2.4.2 Controle de concorrência.....24
2.4.3 Indexação.....24
2.5 Sistemas de Bancos de Dados NoSQL.....25
2.5.1 Características dos bancos NoSQL.....26
2.5.1.1 Escalabilidade horizontal.....26
2.5.1.2 Alta disponibilidade.....27
2.5.1.3 Ausência de esquema ou esquema flexível.....27
2.5.1.4 Acesso a dados via API.....28
2.5.1.5 Consistência eventual.....28
2.5.1.6 Map/Reduce.....28
2.5.1.7 Consistent hashing.....29
2.5.1.8 MVCC.....30
2.5.2 Tipos de banco de dados NoSQL.....30
2.5.2.1 Orientado a colunas.....30
2.5.2.2 Orientado a documentos.....33
2.5.2.3 Armazéns chave/valor.....34
2.5.2.4 Orientado a grafos.....35

3 TRABALHOS RELACIONADOS....36
3.1 Análise da Estrutura do Banco de Dados MongoDB: Teste de Desempenho MongoDB x MySQL....36
3.2 Comparação de Performance entre PostgreSQL e MongoDB....39

4 MATERIAIS E MÉTODOS....41
4.1 Metodologia....41
4.2 Visão geral....42
4.3 Tecnologias utilizadas....42
4.3.1 ISO 2709....43
4.3.2 JSON....43
4.3.3 PostgreSQL....44
4.3.3.1 Limitações do PostgreSQL....46
4.3.4 MongoDB....46
4.3.4.1 Indexação no MongoDB....48
4.3.4.2 Agregação....49
4.3.5 Outras tecnologias....50
4.4 Levantamento de requisitos....51
4.4.1 Requisitos funcionais....51
4.4.2 Requisitos não funcionais....53
4.5 Modelagem da base de dados....53
4.6 Protótipo....57
4.7 Metodologia de teste....60
4.7.1 Teste de busca....60
4.7.2 Teste de carga....61
4.7.3 Teste de volume....61
4.7.4 Teste de Stress....61
4.7.5 Teste de tolerância a falhas....62
4.8 Método estatístico....62
4.9 Ambiente de teste....62
4.9.1 Componentes de hardware e software....63
4.9.2 Versões e configurações dos SGDB's....63

5 RESULTADOS E DISCUSSÃO....64
5.1 Teste de busca....64
5.2 Teste de carga....69
5.3 Teste de volume....75
5.4 Teste de stress....76
5.5 Teste de tolerância a falhas....78
5.6 Impedância....80
5.6.1 Impedância na estrutura....80
5.6.2 Impedância no código fonte....81

6 CONSIDERAÇÕES FINAIS....82

REFERÊNCIAS....85

APÊNDICES....91

1 INTRODUÇÃO

O processo de automação de bibliotecas é indispensável para o atendimento de demandas informacionais, principalmente em bibliotecas universitárias, pois os usuários necessitam ter acesso à informação de diversas áreas. O uso de *softwares* para o gerenciamento de acervos bibliográficos é um fator decisivo no desempenho das funções da biblioteca. A utilização de catálogos eletrônicos diminui o tempo destinado para a realização de serviços inerentes à ela, permitindo ao bibliotecário dedicar-se à outras ações, como o estudo de usuários e outras melhorias demandadas pelo grande volume de informação gerado atualmente. Segundo Barbosa e Eduvirges (2010), a informatização do catálogo do acervo permite que a informação seja recuperada mais rapidamente, atraindo usuários para a biblioteca, além de reduzir o trabalho e custos internos.

Para Furrie (2003), um computador não é capaz de gerar automaticamente um catálogo informatizado de um registro bibliográfico, pois é necessário que as informações sejam interpretadas de alguma forma. Segundo o autor, para que isso ocorra, os dados devem ser armazenados seguindo algumas regras para ser passível de recuperação.

Em 1960 foi criado o padrão de metadados MARC para armazenar registros bibliográficos de diferentes tipos de objetos, com o objetivo de automatizar o processo de catalogação. Esse fato foi um marco que representou a transição da catalogação em fichas para uma catalogação em computador, exercendo profunda influência da disseminação da informação.

Um registro bibliográfico que apresenta o formato MARC possui dados encapsulados em campos e subcampos, que podem ser entendidos como *tags*. Cada *tag* tem o objetivo de

organizar e gerir a informação, além de permitir a sua recuperação. Portanto, diferentes tipos de objetos não serão catalogados da mesma forma, não contendo os mesmos conjuntos de campos e subcampos, uma vez que possuem estrutura diferente. Por se tratarem de registros de estrutura dinâmica, o armazenamento de tais informações não se adapta tão facilmente ao modelo relacional. O formato relacional exige a criação de uma tabela para cada tipo de objeto ou o armazenamento de um registro para cada conjunto de campo e subcampo do registro bibliográfico, que, além de consumir recursos, poderá onerar a busca, devido a grande quantidade de registros para o mesmo registro bibliográfico.

Ao longo dos anos a necessidade de armazenar dados com estruturas flexíveis tem crescido, principalmente em função das aplicações web. De acordo com Lóscio, Oliveira e Pontes (2011), após o advento da web surgiu a necessidade de armazenamento e manipulação de dados semiestruturados e de estrutura dinâmica da internet, que não são suportados por bancos de dados relacionais. Para atender essa necessidade, foram criados os bancos NoSQL (Not Only SQL - Não somente SQL), que possuem como principal característica armazenar dados com estrutura flexível e tratamento de grandes volumes de dados.

O fato de registros bibliográficos em MARC serem dinâmicos e semiestruturados possibilita que os mesmos possam ser armazenados e processados em bancos NoSQL. Dependendo do modelo utilizado, registros inteiros podem ser armazenados como um único registro, sem que exista a necessidade de processar a informação, apresentando vantagem em comparação aos bancos relacionais.

Diante esse contexto, este trabalho visa apresentar um estudo comparativo para o armazenamento e processamento de registros bibliográficos MARC no banco de dados relacional PostgreSQL e NoSQL MongoDB, com ênfase na performance de busca, espaço de armazenamento utilizado, estrutura do banco de dados, integridade dos dados e controle de transação. A fim de obter resultados para o estudo, foi desenvolvido um protótipo para catalogação e busca de registros bibliográficos, simulando um sistema de gestão de acervo de bibliotecas. O protótipo é compatível com ambos bancos de dados, para que os mesmos sejam analisados utilizando a mesma tecnologia na aplicação.

1.1 Motivação

Com maior frequência profissionais de TI estão avaliando a possibilidade de utilizar bancos de dados NoSQL em situações que antes eram atendidas por bancos relacionais. Essa troca está relacionada às características presentes nesses sistemas, como armazenamento de dados semiestruturados, fácil distribuição dos dados, alta disponibilidade e acesso via API's.

Objetos catalogados em registros bibliográficos MARC apresentam campos diferentes entre si, caracterizando-se um registro dinâmico. Essa situação pode ser atendida por bancos de dados relacionais, de forma que cada tipo de objeto seja armazenado em uma tabela própria ou em múltiplas linhas para o mesmo registro, consumindo espaço desnecessário e onerando a sua busca. No entanto, bancos relacionais possuem a vantagem de garantir a consistência dos dados e fazer o controle de transação. Os bancos NoSQL, dependendo do modelo utilizado, podem armazenar um registro bibliográfico como um único registro, e ainda permitem que os relacionados sejam vinculados em um único registro, facilitando a implementação do sistema de gestão de acervos bibliográficos.

Os *softwares* consolidados disponíveis para gerenciar acervos bibliográficos utilizam banco de dados relacionais em sua maioria. Entre eles, podem ser citados o Pergamum (SQL Server e Oracle), Sophia (SQL Server e Oracle) e Gnuteca (PostgreSQL). O Gnuteca está sob licença de *software* livre e armazena os dados bibliográficos no padrão de metadados MARC, possibilitando que o mesmo seja instalado e modificado sem a aquisição de licença. Analisando o *software*, foi constatado que o mesmo é capaz de armazenar dados de qualquer objeto, pois o mesmo é passível de configuração de tipo e tipo físico, e ainda permite que formulários de catalogação específicos sejam criados, escolhendo quais campos MARC o objeto terá. O armazenamento de registros dinâmicos com estrutura flexível é possível devido a estrutura da base de dados, que armazena em uma tabela principal uma linha para cada metadado MARC. Analisada essa estrutura, é possível perceber que é muito custoso para o banco de dados relacional recuperar um registro bibliográfico catalogado, pois é necessário buscar informações em muitos registros.

Tendo como base o contexto citado acima, este estudo visa responder qual paradigma de banco de dados apresenta melhor performance e compatibilidade para busca e armazenamento de registros bibliográficos documentados no formato de metadados MARC, podendo futuramente ser utilizado no *software* Gnuteca.

1.2 Objetivos

Este estudo tem por objetivo geral apresentar uma análise comparativa dos modelos de bancos de dados relacionais e NoSQL quanto ao desempenho e flexibilidade na busca e armazenamento de registros bibliográficos MARC.

1.2.1 Objetivos específicos

Para atingir o objetivo principal definido, os seguintes objetivos específicos deverão ser cumpridos:

a) compreender as principais características do padrão de metadados MARC;

b) compreender as principais características dos bancos relacionais e NoSQL;

c) desenvolver um protótipo para utilização de ambos paradigmas de banco de dados;

d) executar testes de tolerância a falhas, busca, carga, stress e volume dos bancos de dados através do protótipo;

e) avaliar qual modelo atende o armazenamento e busca de metadados para sistemas de gestão de acervos, quanto ao seu desempenho, flexibilidade e facilidade de implementação.

1.3 Organização do trabalho

Com objetivo de compreender este estudo, os capítulos serão apresentados na ordem descrita a seguir.

O capítulo 2 apresenta uma revisão bibliográfica sobre conceitos necessários para o desenvolvimento da proposta, como: metadados, MARC, bancos de dados relacionais e bancos NoSQL.

No capítulo 3, são apresentados estudos semelhantes realizados pela comunidade acadêmica, estudos de casos reais, utilizando bancos de dados relacionais e não-relacionais.

O capítulo 4 apresenta os materiais e métodos utilizados no desenvolvimento do protótipo. O capítulo contém as principais tecnologias que foram utilizadas na elaboração do trabalho, onde serão destacados os bancos de dados PostgreSQL e MongoDB, também são apresentados a metodologia dos testes, o método estatístico e o ambiente onde os testes foram executados.

O capítulo 5 contempla os resultados obtidos na execução dos testes e uma análise qualitativa dos SGDB's em relação a estrutura e desenvolvimento do protótipo.

Por fim, o capítulo 6 apresenta as considerações finais obtidas neste estudo.

2 REFERENCIAL TEÓRICO

Neste capítulo serão abordados os principais conceitos necessários para a análise comparativa de busca e armazenamento entre os SGDB PostgreSQL e MongoDB em metadados MARC. Foram realizados estudos em livros, artigos, monografias e teses. Dentre as informações levantadas, destacam-se conceitos sobre metadados, MARC, características dos Sistemas de Bancos de Dados relacionais e NoSQL.

2.1 Dado

Segundo VAZ (2000), dados podem ser entendidos como fatos relativos à conceitos, objetos, eventos, lugares ou pessoas, eles são abstrações com valor e são armazenados em algum dispositivo, como um documento ou banco de dados. Eles estão associados a: valor, identificação (quais dados são úteis) e tempo (para recuperar o valor).

Para Setzer (1999), dados são uma sequência de símbolos quantificados ou quantificáveis. Logo, um texto é um dado, pois as letras são símbolos quantificados, da mesma maneira que uma imagem, som e animação também são dados.

2.2 Metadado

Segundo Vaz (1999) e Shelley e Johnson (1995, apud ALMEIDA, 1999), metadados são basicamente dados que possuem a função de documentar, avaliar, localizar e selecionar

objetos. Eles fornecem informação que dão características à eles, quanto a sua existência, formato, conteúdo e mídia de intercâmbio.

Para Gil-Leiva (2007), os metadados descrevem e ordenam a informação contida em um documento que pode ser entendido como um objeto. São informações definidoras da descrição formal e análise de conteúdo. Além disso, consistem em estruturas de organização da informação, que podem ser lidas por máquina com o objetivo de tornar os dados úteis.

Os metadados têm a função de definir, descrever e identificar um recurso, com o propósito de filtrar o acesso. São dados que fornecem a documentação de outros, dentro de um ambiente, podendo conter as condições, características e descrição dos dados. Seu uso é importante na recuperação, gestão e organização de informação digital. Exemplos de metadados podem ser assunto, título e autor de um livro, pois eles são utilizados para descrever um livro ou um catálogo de biblioteca. Eles complementam os serviços ou objetos, agregando valor e aumentando seu potencial informativo (ALVES; SOUZA, 2007).

Segundo Almeida (1999), os metadados devem seguir as seguintes regras:

a) fornecer conhecimento quanto ao processamento do conjunto de dados;

b) fornecer conhecimento quanto ao acesso a um conjunto de dados;

c) fornecer conhecimento para determinar quais conjuntos de dados são pertinentes a um assunto em particular.

Para Almeida (1999) os metadados podem ser utilizados para:

a) descrever dados obtidos através de uma fonte externa, fornecendo as informações necessárias para seu processamento;

b) catalogar e disseminar dados produzidos pelas instituições, fornecendo informações sobre eles.

NISO (2004) define os metadados como informações estruturadas, utilizadas para gerenciar e recuperar informação de documentos. O autor destaca três tipos:

a) metadados estruturais: definem a forma de como objetos compostos são estruturados;

b) metadados descritivos: descrevem um recurso para auxiliar na identificação e recuperação. Podem incluir dados como título, palavras-chave e autor;

c) metadados administrativos: oferecem informações que auxiliam no gerenciamento de um recurso. São utilizados para garantir a existência e o acesso no futuro. Exemplos dessas informações são o tipo de arquivo, como foi criado e outras informações técnicas. Eles ainda podem ser divididos em metadados para preservação (fornecem informações necessárias para preservação e armazenagem de um recurso) e metadados para gerenciamento de direitos (guardam informações de gestão de direitos de propriedade intelectual do recurso).

Segundo Gil-Leiva (2007), os metadados utilizam etiquetas ou marcas que são pares de acrônimos ou palavras iguais. A primeira etiqueta marca o início da informação, e a segunda contém uma barra oblíqua e indica o fim da informação. Essas marcações podem facilmente ser lidas por humanos ou programas de computador. Segundo o autor, em 1986 surgiram linguagens de marcação, a partir do conjunto de regras SGML com objetivo de codificar documentos através do uso de etiquetas. Um exemplo de linguagem de marcação é apresentado na Figura 1.

Figura 1 – Linguagem de marcação

```
<nome> Antonio Gil Cuenca </nome>
<lugar de nascimento> Águilas </lugar de nascimento>
<cidade> Murcia </cidade>
<endereço> Rua San Vicence, 7 </endereço>
<parentesco> bisavô paterno </parentesco>
```

Fonte: Gil-Leiva (2007, p. 50)

A partir do SGML foi criada a linguagem XML, um padrão mais extenso e legível por humanos e computadores. Desses modelos, foram criadas novas linguagens de marcação, cada uma com um objetivo específico, seja para distribuir ou esquematizar informação (linguagem HTML) para arquivos de áudio MP3 (ID3), para arquivos audiovisuais (MDL) e para catalogação de acervo bibliográfico (MARC).

Segundo Milstead e Feldman (1999, apud ROCHA, 2004), bibliotecários produzem e padronizam metadados há séculos para indexar e catalogar informações bibliográficas de documentos, com o propósito de fornecer o acesso para o usuário encontrar tais documentos

quando necessário. Segundo o autor, o padrão MARC é um exemplo de uso de metadados para informações bibliográficas, pois eles indicam características de documentos como assunto, título, autor e data de publicação.

Na próxima subseção será apresentado o conceito e característica do padrão de metadados MARC, padrão comumente utilizado para catalogação de acervos bibliográficos, e também o padrão de metadados escolhido para o desenvolvimento deste trabalho.

2.2.1 MARC

Gil-Leiva (2007) e Furrie (2003), definem o padrão MARC como um conjunto de números, letras e símbolos combinados e adicionados aos registros catalográficos. Segundo os autores, o padrão de metadados MARC foi desenvolvido em 1960 pela Biblioteca do Congresso dos Estados Unidos, com objetivo de substituir as fichas catalográficas, e foi originalmente denominado *Library of Congress MARC* (LC MARC).

Segundo Furrie (2003, apud VETTER; ARAUJO, 2012), o MARC foi criado para armazenar informações bibliográficas de diferentes tipos de objetos em computadores com objetivo de automatizar o processo técnico. Esse fato é um marco que representa a transição da catalogação em fichas para uma catalogação realizada por meio de computador. Essa transição exerceu profunda influência na disseminação da informação, que segundo o autor está acontecendo em diferentes formas, de maneira impressa, vídeo, sonora ou digital.

De acordo com Furrie (2003), MARC 21 e MARC podem ser referenciados como sendo o mesmo padrão, pois segundo o autor, MARC 21 é o termo utilizado para denotar o padrão MARC que uniu as diferenças entre os padrões *Canadian Machine-Readable Cataloging* (CAN/MARC) e *United States Machine-Readable Cataloging* (USMARC).

Alves e Souza (2007), definem MARC 21 como um conjunto de padrões utilizados para comunicar, armazenar e identificar informações de acervos bibliográficos em formato legível por máquina para que diferentes programas de computadores possam processar e reconhecer elementos que compõem a informação.

De acordo com Furrie (2003, apud ALVES; SOUZA, 2007), o MARC possibilita a descrição bibliográfica de tipos diferentes de documentos como: *softwares*, mapas, arquivo de

computador, música, vídeo, monografia, periódicos e artigos. Segundo os autores isso é possível pois o padrão MARC utiliza campos fixos e variáveis, subcampos e indicadores. Cada registro é logicamente dividido em campos (*tags*), como título e autor. Cada campo é definido como uma etiqueta de três dígitos numéricos, conforme Figura 2, que podem ser divididos em subcampos e indicadores. O subcampo é representado por um caractere, e tem o objetivo de subdividir o campo em especialidades. Cada *tag* pode ser relacionada com até dois indicadores, esses são representados com dígitos numéricos de 0 a 9 e têm o objetivo de definir alguma informação sobre o campo.

Figura 2 – Campos básicos do formato MARC 21

Campos	**Descrição**
0XX	Informações de controle, números e códigos
1XX	Entrada principal
2XX	Títulos, edição, impressão (em geral, o título, a declaração de responsabilidade, edição e publicação de informação)
3XX	Descrição física, etc.
4XX	Demonstrações, séries
5XX	Notas
6XX	Entradas de assunto
7XX	Entradas secundárias (nome pessoal, entidade, evento, título)
8XX	Entradas secundárias de série
9XX	Uso local

Fonte: Do autor, adaptado de Furrie (2003, apud ALVES; SOUZA, 2007).

Segundo Furrie (2003, apud VALMORBIDA, 2011), os campos de 0XX a 8XX são definidos pelo padrão MARC 21, já os campos 9XX, 09X e 59X são de uso local, podendo ser utilizados conforme a necessidade do utilizador. É importante ressaltar que a notação XX indica que é um conjunto de campos, portanto 2XX engloba *tags* pertencentes ao conjunto de campos: 240, 245, 250, 260, entre outros.

Figura 3 – Registro catalográfico do livro “A assustadora história da maldade”

```
1
2  020 ## $a 8500009853 : $c R$ 45,90
3  100 1# $a Thomson, Oliver.
4  245 12 $a A assustadora história da maldade / $c Oliver Thomson, tradução, Mauro
   Silva.
5  250 ## #a 2. ed.
6  260 ## $a Rio de Janeiro : $b Ediouro, $c c2002.
7  300 ## $a 592 p. $b il. ; $c 24cm.
8  500 ## $a Tradução de: A history of sin.
```

Fonte: Ribeiro (2006, p. 35).

Conforme exemplo da Figura 3, o registro bibliográfico apresenta os elementos de campo, indicador e subcampo. Analisando a primeira linha, “020 “ indica o campo e “##” indica que o campo não possui indicadores. Em seguida, é apresentado o subcampo “a” que representa o ISBN e também o subcampo “c”, que representa o valor.

2.3 Sistemas de banco de dados

Segundo Date (1990), um sistema de banco de dados pode ser entendido como um sistema que tem o objetivo de manter, armazenar e disponibilizar informações quando solicitadas. O banco de dados, como um conjunto de tabelas com informações, é composta por linhas e colunas. Os dados da tabelas estão contidos nas linhas, e as colunas dão o contexto da informação. Essa intersecção é denominada "registro". A tabela deve possuir uma coluna ou um conjunto de colunas que tem o objetivo de identificar o registro, que por sua vez é denominada chave primária.

Para Ramakrishan e Gehrke (2008), um Sistema Gerenciador de Base de Dados se resume em um conjunto de registros ou arquivos, e cada arquivo é um conjunto de uma ou mais páginas. Uma página é uma unidade de gravação ou leitura em disco que pode ter o tamanho parametrizável de 4KB ou 8KB. A organização de arquivos é realizada pelo SGDB, ela deve ser feita de forma organizada, a fim de prover acesso rápido aos dados desejados.

Os primeiros sistemas gerenciadores de banco de dados surgiram em 1960, através de pesquisas patrocinadas por empresas que descobriram que era muito caro manter um grande volume de pessoas para armazenar e organizar arquivos (DIANA; GEROSA, 2010).

Segundo Lóscio, Oliveira e Pontes (2011), os SGDB devem possuir características como: controle de concorrência, segurança, gerenciamento do mecanismo do armazenamento de dados, recuperação de falhas e controle das restrições de integridade do banco de dados.

Nas próximas seções serão apresentados os principais conceitos do modelo de banco de dados relacional e NoSQL.

2.4 Modelo relacional

Segundo Elmasri e Navathe (2005), o banco de dados pode ser representado pelo modelo relacional como uma coleção de relações. A relação, segundo os autores, pode ser pensada como uma tabela de valores ou como um arquivo. Na tabela de valores, a linha é uma coleção de valores de dados relacionados. Cada linha corresponde a uma entidade ou relacionamento no mundo real. A linha será composta por colunas nomeadas para dar significado à informação nelas contidas. As linhas são chamadas de tuplas, um cabeçalho de coluna pode ser chamado de atributo, e a tabela de relação.

Para Ramakrishnan e Gehrke (2008), a relação representa um conjunto de dados, que por sua vez são descritos em um esquema. Segundo o autor, o esquema especifica o nome da coluna e o tipo de cada campo. Para melhor compreensão, é apresentado o exemplo da Figura 4, onde a relação "Alunos" possui os campos "id-aluno", "nome" e "login" do tipo texto onde serão armazenados o código do aluno, nome e o usuário de *login,* respectivamente. Também é apresentado o campo "idade" do tipo número inteiro, onde é armazenada a idade do aluno e o campo "media" do tipo numérico, onde é armazenada a nota final do aluno no semestre.

Figura 4 – Relação Alunos

```
1   Alunos (id-aluno: string, nome: string, login: string, idade: integer, média:
    real)
```

Fonte: Do autor, adaptado de Ramakrishnan e Gehrke (2008).

Silberschatz, Korth e Sudarshan (1999), definem o modelo relacional a partir dos seguintes conceitos:

a) entidade: uma entidade é definida por propriedades, é abstração de objetos do mundo real e tem o objetivo de manter informações;

b) atributo: são as propriedades utilizadas para descrever uma entidade. Os atributos podem ser classificados em simples (único valor para cada entidade), composto (formado por um ou mais sub-atributos), multivalorados (mais de um valor para uma entidade) e derivados (valor de atríbuto derivado de outro);

c) chave: conceito no modelo relacional que permite identificar uma entidade em um conjunto de entidades. As chaves podem ser classificadas em chave primária (representa uma entidade dentro de uma tabela através de uma ou mais colunas) e chave estrangeira (um ou mais campos que representam uma chave primária e permitem o relacionamento).

Nas próximas subseções serão apresentados os conceitos de transações, controle de concorrência e índices.

2.4.1 Transações

Para Silberchatz, Korth e Sudarshan (1999), ações de escrita e leitura são ações/transações únicas de ponto de vista do usuário, portanto, podem ser definidas como uma unidade lógica de execução de programas que acessa ou atualiza vários itens de dados. Para garantir a integridade desses, o SGDB deve manter quatro propriedades denominadas ACID:

a) atomicidade: define a transação de forma atômica, transações se tornam indivisíveis. Ou todas transações são executadas com sucesso ou nenhuma é executada;

b) consistência: assegura que se as modificações forem consistentes, essas serão efetuadas no banco de dados. Caso contrário a transação é cancelada;

c) isolamento: garante que a transação seja executada de forma isolada. Essa propriedade não define qual transação será executada por primeiro, porém garante que uma transação não interfira na outra;

d) durabilidade: garante que uma transação que foi executada com sucesso não seja perdida. Todas transações que tiveram as operações executadas com sucesso até o final serão gravadas permanentemente no banco de dados.

2.4.2 Controle de concorrência

Segundo Elmasri e Navathe (2005), as técnicas de controle de concorrência são necessárias para garantir o isolamento de transações que são executadas simultâneamente/concorrentemente. Elas são implementadas utilizando regras (protocolos). Alguns protocolos utilizam a técnica de bloqueio, que é dividida entre bloqueio binário e compartilhado/exclusivo:

a) bloqueio binário: possui dois estados ou valores, que são bloqueios ou desbloqueios, 0 ou 1 respectivamente. Esse bloqueio é associado a cada item do banco de dados, logo, se o valor de bloqueio é 1, esse estará bloqueado e não poderá ser acessado. Caso for 0 ele estará desbloqueado e poderá ser acessado;

b) bloqueio compartilhado/exclusivo: é um bloqueio multi-modo, permitindo três operações de bloqueio, o modo desbloqueado, o bloqueado-compartilhado (*sharedlocked*) o qual permite que diversas transações leiam o item/dado e o modo bloqueado-exclusivo (*exclusive-locked*), que permite apenas uma transação para controlar o bloqueio.

Os SGDB relacionais permitem a conversão entre os modos de bloqueio, e são utilizadas técnicas para prevenir que ocorra o *deadlock*. Alguns protocolos utilizam *timestamps* (marcas de tempo) para cada transação, que garantem o controle através da ordenação das transações. Existem protocolos multiversão, que garantem múltiplas versões para um determinado item/dado e existem também protocolos baseados no conceito de certificação/validação da transação, ao final de suas operações.

2.4.3 Indexação

De acordo com Ramakrishan e Gehrke (2008), a técnica de indexação pode ser utilizada no acesso aos dados/arquivos, fazendo-o de forma eficaz, otimizando o acesso ao disco. Um índice pode ser entendido como um arquivo que auxilia uma tabela, possuindo ponteiros para colunas específicas. Eles podem ser primários ou secundários. O primário considera colunas que incluam a chave primária, já os secundários são todos os outros.

A indexação pode ser feita baseada em *hash* ou em árvore. A indexação em *hash* pode encontrar rapidamente registros que possuam valor na chave de pesquisa. Esse tipo de indexação permite incluir, excluir e buscar dados com até uma operação de entrada/saída. Já na indexação em árvore, os dados são organizados de forma hierárquica e de maneira ordenada pelo valor da chave de pesquisa. Cada nó da árvore é uma página física, sendo o menor nível da árvore denominada de folha, que é onde estão os dados. Essa estrutura permite localizar os registros de forma muito eficiente, pois todas pesquisas iniciam no nó raiz, e os nós direcionam as pesquisas para as folhas corretas.

2.5 Sistemas de Bancos de Dados NoSQL

Segundo Lóscio, Oliveira e Pontes (2011), a evolução das aplicações de banco de dados demandou o armazenamento de formatos de dados não suportados pelos bancos relacionais, como imagem, som e vídeo e tipos de dados complexos. Para atender essas novas necessidades, foram propostos os bancos orientados a objetos e os bancos de dados objeto relacionais, também conhecidos como BDOO e BDOR, respectivamente. Após o surgimento da web, novas necessidades surgiram, como o armazenamento e manipulação de grandes volumes de dados semi-estruturados e não estruturados. Para atender essa nova demanda foram criados os bancos NoSQL.

Para Diana e Gerosa (2010), embora o termo NoSQL só tenha surgido em 2009, desde o surgimento da Web 2.0 empresas criaram suas próprias soluções para escalabilidade no armazenamento e processamento de grandes volumes de dados e publicaram essas soluções em artigos científicos. Segundo os autores, o termo surgiu quando a comunidade de *software* livre e empresas da web iniciaram o desenvolvimento de seus próprios bancos de dados não-relacionais baseados naqueles artigos.

Segundo Leavitt (2010, apud FUHR, 2014), o fator para o surgimento dos bancos NoSQL foi o crescimento do uso dos sistemas computacionais nas empresas, que atualmente demandam o armazenamento de dados no nível de petabytes (10^{15} *bytes*). A medida que esse volume de dados aumenta, duas soluções podem ser utilizadas: aumentar o número de servidores de banco de dados ou aumentar a capacidade do servidor. Quando essas alternativas não são possíveis, a solução é distribuir o banco de dados. Segundo o autor, esse

cenário não pode ser atendido de maneira simples por bancos de dados relacionais, pois, devido a sua natureza estruturada, a distribuição dos dados é complexa.

NoSQL (2010, apud DIANA; GEROSA, 2010), define o termo NoSQL como impreciso e confuso, pois o termo SQL não significa banco de dados relacional e suas limitações. Por alguns bancos dessa categoria utilizarem SQL, essa categoria de bancos de dados pode ser definida como “Não apenas SQL”. Segundo o autor, esses bancos apresentam a maioria das seguintes características: ausência de esquema flexível, alta disponibilidade, não-relacional, escalável horizontalmente, código aberto, com suporte nativo à replicação de dados e acesso via API's.

Segundo Chang et al. (2010, apud FUHR, 2014), os bancos de dados NoSQL devem ser utilizados nos casos em que exista a necessidade de maior flexibilidade do banco de dados, portanto, o seu objetivo não é substituir o modelo relacional.

Nas próximas subseções serão apresentadas as principais características dos bancos de dados NoSQL, algumas técnicas importantes para a implementação de suas funcionalidades, modelos de bancos de dados e implementações existentes.

2.5.1 Características dos bancos NoSQL

Nesta subseção serão apresentadas as principais características que diferenciam os sistemas de bancos de dados NoSQL dos relacionais.

2.5.1.1 Escalabilidade horizontal

Tiwari (2011) define a escalabilidade como a possibilidade do sistema aumentar sua capacidade através da adição de recursos para tratar o aumento de sua carga.

Segundo Lóscio, Oliveira e Pontes (2011), o crescimento de volume de dados requer a melhoria de desempenho dos bancos de dados e de escalabilidade. Nesse cenário, existem duas soluções possíveis, a escalabilidade vertical (aumentar o poder de processamento e armazenamento do servidor) e a escalabilidade horizontal (aumentar e distribuir processos do banco de dados). A ausência de bloqueios é uma característica dos bancos de dados NoSQL

que torna a escalabilidade horizontal uma solução mais viável, pois ela soluciona o problema de gerenciamento de grandes volumes de dados. Segundo os autores, essa tecnologia não pode ser utilizada em bancos de dados relacionais, pois processos executados simultaneamente e concorrendo entre si podem aumentar o tempo de acesso às tabelas.

Na mesma linha de pensamento de Lóscio, Oliveira e Pontes (2011), Tiwari (2011) define que o aumento de processamento e armazenamento de grandes volumes de dados pode ser atendido pela escalabilidade vertical e horizontal. Para o autor citado, a vertical geralmente é cara e de tecnologia proprietária, e a escalabilidade horizontal pode se tornar complexa, necessitando o uso do modelo *Map/Reduce* para o processamento de dados em larga escala em um *cluster* de máquinas.

2.5.1.2 Alta disponibilidade

Para Tiwari (2011), alta disponibilidade significa que o sistema deve estar disponível para servir sempre que for necessário. Quando o sistema estiver ocupado, pouco comunicativo ou não responder, o sistema é considerado indisponível.

Moniruzzaman e Hossain (2013) definem alta disponibilidade como a característica do banco de dados que permite que clientes sempre encontrem ao menos uma cópia de seus dados quando solicitado.

2.5.1.3 Ausência de esquema ou esquema flexível

Segundo Lóscio, Oliveira e Pontes (2011), bancos de dados NoSQL apresentam a característica de ter total ou quase total ausência de um esquema de definição de estrutura de metadados. Essa característica proporciona um aumento na disponibilidade e na escalabilidade do banco de dados, em contrapartida, não garante a integridade dos dados. Segundo os autores, esse modelo apresenta flexibilidade na organização dos dados e geralmente são baseados no conceito chave-valor.

2.5.1.4 Acesso a dados via API

Para Lóscio, Oliveira e Pontes (2011), soluções NoSQL devem ser de alta disponibilidade, escaláveis e proverem uma forma fácil para que aplicações possam utilizar os dados de forma rápida e eficiente. Segundo os autores, essa característica pode ser obtida através do uso de API's que facilitam o acesso às informações. Dessa maneira, não é necessário se preocupar com a forma de armazenamento dos dados.

2.5.1.5 Consistência eventual

Lóscio, Oliveira e Pontes (2011) definem que a consistência eventual é uma característica em soluções NoSQL e tem como princípio o teorema de CAP. Segundo os autores, ele define que não é possível ter as três propriedades no banco de dados simultaneamente. Dessa maneira, as propriedades ACID não podem ser respeitadas em sua totalidade pois bancos de dados NoSQL priorizam a disponibilidade e tolerância a falhas.

2.5.1.6 Map/Reduce

Segundo Lóscio, Oliveira e Pontes (2011), m*ap/reduce* é um modelo de programação criado pelo Google e é utilizado para processar grandes volumes de dados que estão distribuídos em *clusters* de computadores. O modelo é implementado através de duas funções:

a) *map*: os problemas são quebrados em subproblemas e são distribuídos nos nós participantes do *cluster*. A função processa um par chave/valor, gerando um conjunto de pares chave/valor;

b) *reduce*: os subproblemas que estão sendo executados nos nós filhos são resolvidos e repassados ao pai. No nó pai, que também pode ser um nó filho, os resultados que possuem mesma chave são mesclados e repassados ao seu pai. Esse processo ocorre até o resultado chegar ao nó raiz do problema.

Para exemplificar, na Figura 5 é apresentado um exemplo das funções *map* e *reduce* no banco de dados MongoDB. No exemplo são inseridos *urls* repetidas na coleção sites. Logo

após é criada a função “map*()”* e função “reduce()”. A função “map*”* define que dados serão agrupados pelo campo “url”, e a função “reduce()” define que será feita uma contagem de registros. Após a declaração das funções, é aplicado um “mapReduce()*”* na coleção sites, cujo retorno será a coleção “mapped_urls*”*.

Figura 5 – *Map/Reduce* na coleção sites

```
2  // Inserção de dados na coleção sites.
3  db.sites.insert({url: "www.google.com", date: new Date(), trash_data: 5});
4  db.sites.insert({url: "www.loc.gov", date: new Date(), trash_data: 13});
5  db.sites.insert({url: "www.google.com", date: new Date(), trash_data: 1});
6  db.sites.insert({url: "www.loc.gov", date: new Date(), trash_data: 69});
7  var map = function() {
8    emit(this.url, 1);
1  }
2  var reduce = function(key, values) {
3     var res = 0;
4     values.forEach(function(v){ res += 1; });
5     return {count: res};
6  }
7  db.sites.mapReduce(map, reduce, { out : "mapped_urls"});
```

Fonte: Elaborado pelo autor (2015).

Após aplicar o “mapReduce” para visualizar os resultados, é necessário aplicar a função “find()” na coleção “mapped_urls”, conforme Figura 6. No exemplo, é possível observar que o resultado apresenta documentos com o campo “value” contendo a quantidade de documentos que possuem determinada *url*.

Figura 6 – Resultado do *MapReduce* na coleção sites

```
1
2  db.mapped_urls.find({});
3  {"url" : "www.google.com.br", "value" : {"count" : 2} }
4  {"url" : "www.loc.gov", "value" : {"count" : 2} }
```

Fonte: Elaborado pelo autor (2015).

2.5.1.7 Consistent hashing

O mecanismo *consistent hashing* pode ser entendido como um anel de objetos (dados) e nós (bancos de dados), assim que determinado nó sai do anel, seus objetos são mapeados para o próximo nó no sentido horário. Esse mecanismo dá suporte para recuperação e armazenamento de dados em bancos distribuídos, onde a quantidade de nós está em constante modificação (LÓSCIO; OLIVEIRA; PONTES, 2011).

2.5.1.8 MVCC

Para Lóscio, Oliveira e Pontes (2011), o MVCC permite que operações de escrita e leitura sejam executadas simultaneamente, pois dá suporte a transações paralelas em bancos de dados sem o uso de bloqueios, como as utilizadas em bancos relacionais. O método de controle de concorrência por versão implementa a atualização por substituição, e não por exclusão do dado antigo. Dessa forma existirão muitas versões do mesmo dado, mas somente a mais recente será válida.

2.5.2 Tipos de banco de dados NoSQL

Para Diana e Gerosa (2010), os tipos mais comuns de bancos de dados NoSQL são: orientado a colunas, orientado a documentos, armazéns de chave-valor e orientado a grafos.

A seguir serão apresentadas algumas características que os diferenciam de bancos de dados relacionais, bem como as vantagens e desvantagens.

2.5.2.1 Orientado a colunas

Segundo Stonebraker et al. (2005, apud DIANA; GEROSA, 2010), bancos de dados relacionais são orientados a linhas, portanto, armazenam registros das tabelas contiguamente no disco, conforme Figura 7. Essa estrutura torna o processo de armazenamento eficaz, pois o processo ocorre em uma única escrita no banco de dados, e é eficaz também para leitura de registros inteiros, pelo fato da linha toda estar armazenada em sequência no disco. No entanto, segundo o autor, essa estrutura é pouco eficaz para leitura de poucas colunas de muitos registros, pois muitos blocos terão de ser lidos. Bancos de dados orientados a colunas armazenam dados das colunas contiguamente, conforme Figura 8, portanto, se torna mais interessante para os casos onde se deseja otimizar a leitura de poucas colunas de muitos registros de dados estruturados.

Figura 7 – Registro de uma tabela banco de dados relacional armazenada em disco

```
1   Código1, Nome1, Endereço1, CEP1, Estado1, Pais1, Telefone1, Email1, Codigo2,
    Nome2, Endereco2, CEP2, Estado2, Telefone2, Email2, Nome3, Endereco3, CEP3,
    Estado3, Pais3, Telefone3, Email3
```

Fonte: Elaborado pelo autor (2015).

Figura 8 – Registro de uma tabela banco de dados orientado a colunas

```
1   Código1, Codigo2, Codigo3, Nome1, Nome2, Nome3, Endereço1, Endereco2, Endereco3,
    CEP1, CEP2, CEP3, Estado1, Estado2, Estado3, Pais1, Pais2, Pais3, Telefone1,
    Telefone2, Telefone3, Email1, Email2, Email3
```

Fonte: Elaborado pelo autor (2015).

Para Lóscio, Oliveira e Pontes (2011), o paradigma orientado a colunas também pode ser chamado de orientado a atributos, onde os dados são indexados em uma tripla (linha, coluna e *timestamp*). Nessa indexação, as linhas e colunas são identificadas por uma chave e o *timestamp* identificará qual é a última versão do valor.

De acordo com Diana e Gerosa (2010), a escrita de um novo registro em uma tabela é mais custosa no paradigma orientado a colunas quando comparado com um banco de dados relacional (orientado a linhas). Portanto, o uso de bancos orientados a colunas são mais adequados para processamento analítico *online* (OLAP), e bancos relacionais são mais adequados para processamento de transações *online* (OLTP).

Os principais SGDB's desse paradigma são Google BigTable, Hypertable, Hbase e Cassandra.

Segundo Chang et al. (2006), o BigTable é um banco orientado a colunas desenvolvido em 2004 pelo Google e foi projetado para escalar *petabytes* de dados, podendo utilizar milhares de máquinas. Suas principais características são: aplicabilidade ampla, escalabilidade, alto desempenho e alta disponibilidade. Segundo os autores, o BigTable é utilizado em produtos que demandam uma grande carga de informações como Google Analytics e Google Earth. O BigTable fornece aos usuários um modelo simples de dados, que suporta controle dinâmico sobre seu *layout* e formato. O BigTable trata os dados como *Strings* não interpretadas e pode armazenar dados na memória RAM ou no disco rígido.

De acordo com Lóscio, Oliveira e Pontes (2011), o BigTable permite escalabilidade de recursos e alta disponibilidade no processamento de dados. É utilizado em mais de 60

produtos do Google, entre eles estão o Gmail e Google Drive. Segundo os autores, o BigTable é utilizado em conjunto com outros pacotes do Google, como o Google File System (GFS) para o processamento de informações e *map*/*reduce* para distribuição dos dados.

O SGDB Cassandra é um projeto da Apache sob a licença Apache 2.0 que foi iniciado no Facebook e foi desenvolvido utilizando conceitos do Dynamo da Amazon e o BigTable do Google. É um banco de dados distribuído que possui a característica de poder gerenciar grandes volumes de dados estruturados através de servidores em *cluster*, funcionando de forma altamente disponível. O Cassandra garante alta disponibilidade pelo fato de utilizar a arquitetura em forma de anel sem um nó mestre ao invés de utilizar a arquitetura mestre-escravo. Nessa arquitetura todos nós possuem a mesma função, todos se comunicam entre si. Dessa forma, não existe nenhum ponto de falha (THE APACHE SOFTWARE FOUNDATION, 2015).

Para Hewitt (2011), o Cassandra é um banco de dados orientado a colunas, de código livre, que possui a característica de parecer como um único processo para o usuário. Segundo o autor, o Cassandra é escalável horizontalmente de forma elástica, podendo aceitar ou remover nós sem impactar no sistema como um todo. Assim que um novo nó é adicionado, ele recebe alguns ou todos os dados e já estará disponível para receber novas requisições.

2.5.2.2 Orientado a documentos

Segundo Tiwari (2011, apud FUHR, 2014), a flexibilidade é a característica que difere o paradigma orientado a documentos do paradigma relacional, pois diferente do conceito de armazenamento em linhas de tabelas, o modelo armazena documentos. Os dados armazenados podem ser entendidos como uma coleção de documentos, o qual cada um contém sua própria estrutura e hierarquia.

Para Diana e Gerosa (2010), os documentos armazenados nos SGDB's orientados a documentos são coleções de atributos e valores. Segundo os autores, geralmente esse paradigma não implementa esquema de definição de dados, portanto, os documentos não precisam seguir uma estrutura imposta, tornando o paradigma adequado para o armazenamento de dados semiestruturados.

De acordo com Lóscio, Oliveira e Pontes (2011), em função do SGDB não apresentar um esquema rígido como dos sistemas relacionais, torna possível atualizar a estrutura de documentos, remover ou adicionar campos, sem causar problemas ao banco de dados.

Formatos comuns de documentos como o JSON são comumente utilizados em paradigmas orientado a documentos, pois eles permitem que um documento seja embutido dentro de outro. Essa técnica favorece a distribuição do sistema, pois dados próximos podem ser um único documento. Uma desvantagem apontada pelos autores é duplicação de dados, que pode criar problemas de consistência, causados por anomalias de remoção e atualização (DIANA; GEROSA, 2010).

O fato do modelo ser indicado para armazenamento de dados semiestruturados, permitir atualização de estrutura sem afetar o funcionamento do sistema e permitir que documentos possam ser embutidos dentro de outros torna o modelo favorável para o armazenamento e processamento de documentos contendo informações bibliográficas de diferentes tipos de obras em uma biblioteca. Por exemplo, um registro de livro irá conter a quantidade de páginas, um registro de filme conterá a duração em minutos. Outro exemplo que pode ser citado é a ligação entre uma coleção de periódico e seus fascículos.

Conforme exemplo da Figura 9, documentos representando fascículos podem ser embutidos em um que representa a coleção.

Figura 9 – Documento JSON contendo dados da coleção de periódicos Super Interessante

```
1  {
2    "numero_controle": "1",
3    "titulo": "Super interessante",
4    "data_publicacacao":"1990",
5    "assunto":["Generalidades","Periódico"],
6    "idioma":"Português",
7    "lugar_publicacao": "São Paulo",
8    "editor": "Abril",
9    "fasciculos":[ {
10       "titulo": "Super interessante",
11       "numero: "7",
12       "ano": "jul. 2003"
13     }]
14 }
```

Fonte: Elaborado pelo autor (2015).

Os principais SGDB's desse paradigma são MongoDB e Apache CouchDB. Segundo Vieira et al. (2012) e The Apache Software Foundation (2015), o CouchDB é um banco de

dados implementado na linguagem Erlang e é utilizado para armazenar documentos no padrão JSON. Possui interface totalmente web e fornece acesso aos documentos no formato de serviço *RestFul* e (HTTP/JSON Api). Tanto as consultas quanto as transformações são realizadas através da linguagem Javascript, no entanto, segundo os autores, também podem combinar as linguagens Pyhthon e CoffeScript, o que o torna ideal para ser utilizado em aplicativos web e aplicativos para dispositivos móveis.

O CouchDB controla a concorrência seguindo a política MVCC e os conflitos e integridade devem ser resolvidos no nível de aplicação. Nativamente o CouchDB não é escalável horizontalmente, para utilizar esse recurso é necessário utilizar o BigCouch, que facilita a elaboração de *cluster* elásticos de instâncias CouchDB (VIEIRA et al., 2012).

2.5.2.3 Armazéns chave/valor

O modelo pode ser entendido como uma grande tabela *hash*, pois o SGDB é composto por um conjunto de chaves, que são associados a um único valor *String* ou binário. Esse modelo permite que os dados sejam alterados e lidos através da chave, pelas funções *set()* e *get(),* respectivamente, o que o torna de fácil implementação. Essa característica contribui para aumentar a disponibilidade dos dados. No exemplo da Figura 10, a chave é um campo como nome e sexo, já o valor é a instância para o campo correspondente.

Figura 10 – Registro de chave/valor

```
Nome: Jader Fiegenbaum
Idade: 25
Sexo: Masculino
Fone: (99) 9999-9999
```

Fonte: Elaborado pelo autor (2015).

Uma desvantagem apresentada nesse modelo é o fato de não permitir o acesso a objetos através de consultas mais complexas (LÓSCIO; OLIVEIRA; PONTES, 2011).

Os principais SGDB's desse paradigma são: Amazon Dynamo, GenieDB, Redis e Riak.

2.5.2.4 Orientado a grafos

Para Lóscio, Oliveira e Pontes (2011), o paradigma de bancos de dados orientados a grafos é composto por três componentes:

a) nós: são os vértices do grafo;

b) relacionamentos: são as arestas que ligam os nós;

c) propriedades: atributos que representam características dos nós e relacionamentos.

O banco de dados pode ser entendido como um multigrafo rotulado e direcionado. Cada par de vértices pode estar conectado por uma ou mais arestas. Os principais SGDB's desse paradigma são: Neo4j, AllegroGraph e Virtuoso.

No próximo capítulo serão apresentadas duas análises comparativas relacionadas a este trabalho. A primeira delas apresenta uma análise de performance entre os SGDB's MongoDB e MySQL e a segunda entre MongoDB e PostgreSQL.

3 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados trabalhos produzidos pela comunidade acadêmica, onde são comparados bancos relacionais e NoSQL. Ambos trabalhos estão relacionados a este estudo, pois o primeiro realizou testes quantitativos de desempenho entre os sistemas MongoDB e MySQL, avaliando a performance em relação ao tempo e espaço em disco através da aplicação de testes de carga, volume e stress. O segundo realiza um análise quantitativa mais simples, onde foi avaliada a performance em relação ao tempo dos sistemas MongoDB e PostgreSQL através de inserção e busca de dados.

3.1 Análise da Estrutura do Banco de Dados MongoDB: Teste de Desempenho MongoDB x MySQL

O trabalho produzido por Rosa (2013), teve como objetivo comparar os bancos de dados MongoDB e MySQL, modelos NoSQL e relacionais, respectivamente, e foi motivado pela crescente procura desse novo paradigma de armazenamento e processamento de dados. O trabalho é contemplado por uma fundamentação teórica sobre o paradigma NoSQL e MongoDB.

Para realizar os testes comparativos, o autor se propôs a criar um protótipo de blog capaz de utilizar ambos os bancos. O blog permite criação de artigos, cadastrar usuários, criar comentários e realizar testes de carga, volume e stress.

O teste de carga foi realizado através da aplicação de dois algoritmos para determinar o grau de aceitabilidade do banco de dados submetido a uma determinada demanda de carga de trabalho.

O primeiro algoritmo inseriu dados fazendo a persistência dos mesmos. O resultado é apresentado na Tabela 1.

Tabela 1 – Resultado do algoritmo 1 em função do tempo e tamanho

Registros	MySQL(s)	MongoDB(s)	MySQL(GB)	MongoDB(GB)
155	4,32	0,14	0,0002290	0,203125
1110	30,51	0,22	0,000641	0,203125
8420	31,03	0,30	0,001447	0,203125
27930	849,65	0,53	0,012848	0,203125
127500	3807,42	1,64	0,054258	0,203125
347970	10505,93	4,79	0,111847	0,453125
518480	16452,92	25,87	0,165726	0,953125
737190	21915,67	22, 37	0,2333687	0,953125
1010100	31991,27	30,45	0,317551	0,953125

Fonte: Do autor, adaptado de Rosa (2013).

O segundo algoritmo de carga aplicado realizou a persistência de múltiplos valores, inserindo dados em blocos ao invés de um a um. O resultado dos testes é apresentado na Tabela 2.

Tabela 2 - Resultado do algoritmo 2 em função do tempo

Registros	MySQL(s)	MongoDB(s)
155	0,203	0,003
1110	0,392	0,013
3615	0,559	0,027
8420	0,632	0,057
27930	1,611	0,187
65640	2,916	0,503
127550	6,009	0,995
219660	12,07	1,752

Fonte: Do autor, adaptado de Rosa (2013).

O teste de volume foi realizado através da aplicação de um algoritmo contendo duas funções, a primeira foi desenvolvida para ser capaz de buscar todos dados contidos na coleção

de dados do MongoDB e a segunda capaz de obter todos os dados de três tabelas no MySQL. O resultado é apresentado na Tabela 3.

Tabela 3 – Resultado do algoritmo de teste de volume

Registros	MySQL(ms)	MongoDB(ms)
1110	3	1
8420	12	1
27930	40	2
65640	94	4
127550	178	6
219660	309	9

Fonte: Do autor, adaptado de Rosa (2013).

O teste de stress foi realizado através da aplicação de um algoritmo capaz de ser executado em múltiplas *threads*, para simular acessos simultâneos ao banco de dados. O resultado é apresentado na Tabela 4.

Tabela 4 - Resultado do algoritmo de teste de stress

Registros	MySQL(s)	MongoDB(s)
10	2,784	0,119
20	6,225	0,194
30	9,274	0,278
40	12,986	0,368
50	15,148	0,437

Fonte: Do autor, adaptado de Rosa (2013).

De acordo com Rosa (2013), o teste de carga determinou o nível de aceitabilidade dos sistemas ao lidar com uma grande carga de trabalho, o teste de volume avaliou o comportamento ao lidar com grande volume de dados, e o teste de stress determinou o nível de aceitabilidade. Segundo o autor, os resultados obtidos através do desenvolvimento de um protótipo foi satisfatório por determinar que o MongoDB é um banco de dados de alta performance em relação ao MySQL.

O trabalho desenvolvido por Rosa (2013) apresenta somente uma análise quantitativa de performance, obtida através da execução de algoritmos programados, onde o usuário não é capaz de alterar variáveis ao executar o teste, como por exemplo a quantidade de registros que

serão analisados. O autor também não explorou as principais diferenças entre os SGDB's analisados e as configurações utilizadas em cada um.

3.2 Comparação de Performance entre PostgreSQL e MongoDB

O trabalho produzido por Politowski e Maran (2014), foi estruturado em seções de trabalhos relacionados e estudos bibliográficos, onde foram apresentadas as principais características dos bancos de dados utilizados, seguido da descrição dos testes efetuados, ambiente utilizado e a apresentação dos resultados obtidos através da aplicação dos testes, e por fim, a conclusão dos autores sobre os resultados obtidos.

A fase de testes foi dividida em três categorias:

a) inserção: adiciona dados simples em uma tabela;

b) busca simples: busca com apenas uma cláusula de filtro em apenas uma tabela;

c) busca complexa: busca em inúmeras tabelas com junções, subconsultas e mais de uma cláusula de filtro.

Cada teste foi executado em laços de 1, 10, 100, 1.000, 10.000 e 100.00 repetições, sendo executados três vezes cada, e o resultado considerado foi a média aritmética das três iterações. Após a execução dos testes, os autores elaboraram uma tabela de resultados, que é apresentada na Tabela 5 para PostgresSQL e Tabela 6 para MongoDB, ambas apresentam os dados em segundos.

Tabela 5 – Performance do PostgreSQL

Operações	**Número de repetições**					
	1	10	100	1000	10000	100000
Inserção	0,0096	0,1108	1,0792	11,0729	105,1221	1106,3959
Busca simples	0,3902	3,9198	39,5233	451,3150	4286,2453	40892,9120
Busca complexa	2,3452	24,4739	250,1055	2568,9904	26929,7949	278438,6800

Fonte: Do autor, adaptado de Politowski e Maran (2014).

Tabela 6 – Performance do MongoDB

Operações	**Número de repetições**					
	1	10	100	1000	10000	100000
Inserção	0,0046	0,0072	0,0781	0,7294	7,0441	77,7849
Busca simples	0,0007	0,0051	0,0528	0,5159	5,0282	49,7979
Busca complexa	0,0008	0,0085	0,0529	0,5068	5,1852	51,7124

Fonte: Do autor, adaptado de Politowski e Maran (2014).

Segundo os autores, a execução de testes de inserção e busca, sem alterar parâmetros de configuração dos sistemas foi suficiente para concluir que o MongoDB foi criado para suprir a necessidade de performance, deixando de lado características presentes em bancos de dados relacionais. Os autores apontam que o MongoDB é recomendado para aplicações que necessitam de grande carga de consultas, como serviços web.

No próximo capítulo serão apresentados os materiais e métodos utilizados para realizar a análise quantitativa e qualitativa entre os SGDB's PostgreSQL e MongoDB para o armazenamento e busca de metadados MARC.

4 MATERIAIS E MÉTODOS

Este capítulo tem como objetivo apresentar a metodologia científica empregada, os artefatos, documentos e técnicas que foram utilizadas no desenvolvimento do estudo.

4.1 Metodologia

Com a finalidade de atender aos objetivos propostos, esta seção apresentará o enquadramento metodológico do estudo.

Segundo Malhotra (2006, apud CHEMIN, 2015), a pesquisa qualitativa tem por objetivo compreender qualitativamente as motivações, as razões de determinado problema e apresenta uma coleta de dados não-estruturados, com análise de dados não-estatísticos. Para Leopardi (2002, apud CHEMIN, 2015), a pesquisa qualitativa é utilizada quando não é possível obter dados precisos do objeto de estudo e quando se deseja obter dados subjetivos. Portanto, segundo Appoinário (2006, apud CHEMIN, 2015), este tipo de pesquisa não busca a generalização, mas sim compreender qualitativamente um fenômeno.

A pesquisa quantitativa contempla o que pode ser mensurado, contado e medido. Ela é utilizada em situações que exigem estudo exploratório do problema e é adequada para quando se deseja conhecer quantitativamente o objeto de estudo (LEOPARDI, 2002; MEZZAROBA; MONTEIRO, 2006). Segundo Brenner e Jesus (2007, apud CHEMIN, 2015) a pesquisa quantitativa apresenta resultados para auxiliar a comparação e análise de dados, geralmente apresentados em forma de gráficos ou tabelas.

A pesquisa realizada neste estudo pode ser enquadrada em quali-quantitativa quanto ao modo de abordagem, pois os SGDB's PostgreSQL e MongoDB serão comparados quantitativamente através dos testes de busca, carga, volume e stress e comparados qualitativamente através do desenvolvimento das rotinas de busca, edição e inserção dos dados. Quanto ao objetivo geral, a pesquisa realizada pode ser enquadrada em pesquisa exploratória, pois segundo Gil (2006), Leopardi (2002) e Malhotra (2006), ela envolve revisão de conceitos na literatura, análise de exemplos e testes padronizados, cujo objetivo é aumentar o conhecimento do pesquisador sobre um determinado problema. A pesquisa também pode ser enquadrada em pesquisa bibliográfica, quanto aos procedimentos técnicos, pois para Chemin (2015), este tipo de pesquisa é desenvolvido com base em obras literárias, livros de referência, publicações periódicas, anais de encontros científicos e materiais encontrados em meios digitais.

4.2 Visão geral

O estudo realizado teve como objetivo descobrir como os modelos de bancos de dados relacionais e NoSQL atendem o armazenamento e busca de metadados MARC, devido a natureza dos dados. Para isso, foi desenvolvido um protótipo para inserir, editar e pesquisar registros bibliográficos nos SGDB's PostgreSQL e MongoDB. Todas as operações executadas tiveram o tempo medido e mostrado em todas as interfaces. O protótipo também contempla teste de busca, tolerância a falhas e testes de carga, stress e volume do banco de dados, com objetivo de avaliar a performance de cada modelo.

Os testes disponíveis contemplam a comparação do PostgreSQL e MongoDB executado em apenas um nó (de maneira não *sharding*).

4.3 Tecnologias utilizadas

Esta seção tem como objetivo apresentar os fundamentos e características das tecnologias que foram utilizadas para a elaboração do estudo, afim de atender os objetivos propostos. Entre as tecnologias está o ISO 2709, formato utilizado para importação de materiais no protótipo, o JSON, tipo de documento que é armazenado no MongoDB e os Sistemas de Banco de Dados PostgreSQL e MongoDB, objetos de estudo deste trabalho.

4.3.1 ISO 2709

Segundo Cortê et al. (1999), a ISO 2709 (*Documentation Format for Bibliographic Interchange on Magnetic Tape*) é uma norma que descreve o formato dos registros bibliográficos para o seu intercâmbio entre sistemas. Ela foi desenvolvida pelo Comitê Técnico ISO/TC 46 da ISO em 1960. De acordo com o autor, a norma foi projetada especificamente para a comunicação entre sistemas e não define significado para o seu conteúdo, essa definição é função do formato de implementação. Neste trabalho, o formato de implementação utilizado é o MARC 21.

4.3.2 JSON

Segundo o manual do JSON (2015), ele é um formato simples para intercâmbio de dados, é fácil de ser lido e entendido por humanos e fácil para ser gerado e interpretado por computadores. O formato foi proposto Douglas Crockford em 1999 como uma alternativa para XML. Ele consiste em um formato texto independente de linguagem de programação e é constituído pelas seguintes estruturas universais:

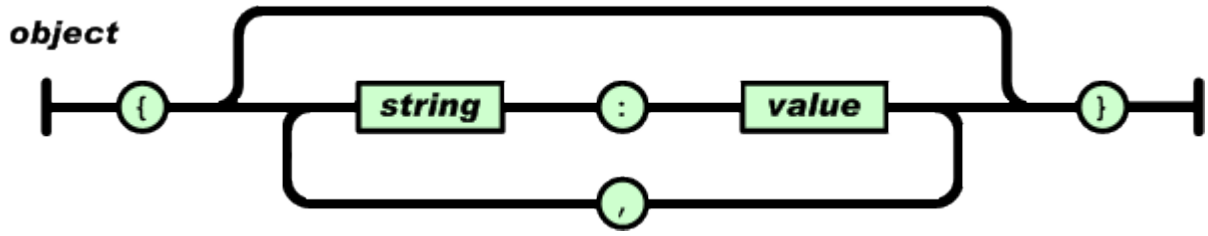
a) coleção de pares nome/valor;

b) lista ordenada de valores;

c) valores.

O formato define a coleção de pares nome/valor como um objeto, cujo caractere inicial é uma chave de abertura "{" e termina com "}", nomes (atributos) são seguidos de dois pontos ":" e objetos são separados por vírgula "," (FIGURA 11).

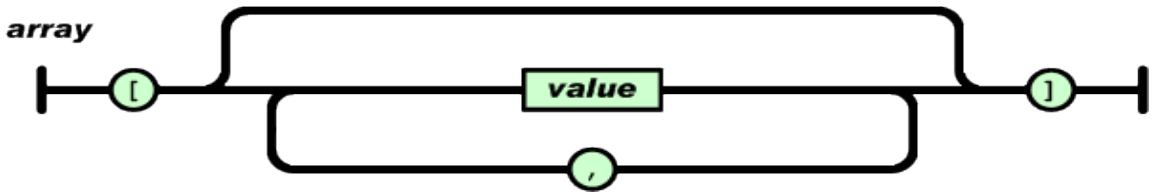
Figura 11 – Estrutura de um objeto JSON



Fonte: Manual do JSON (2015).

Uma lista ordenada de valores é um vetor. Conforme Figura 12, ele inicia com um colchete de abertura “[“ e termina com um colchete de fechamento “]” cujos valores são separados por vírgula “,”.

Figura 12 – Estrutura de um vetor em JSON



Fonte: Manual do JSON (2015).

O valor no JSON pode ser:

a) uma cadeia de caracteres (*String*);

b) valor verdadeiro (*true*);

c) valor falso (*false*);

d) valor nulo (*null*);

e) um número;

f) um vetor (*array*);

g) um objeto JSON (*object*).

4.3.3 PostgreSQL

Segundo o manual do PostgreSQL (2015), ele é um Sistema Gerenciador de Banco de Dados objeto-relacional e *opensource* (código aberto). O projeto foi liderado pelo professor Michael Stonebraker e foi patrocinado pela Army Research Office (ARO) e Defense Advanced Research Projects Agency (DARPA) em 1996, e atualmente está na versão 9.4. Segundo o autor, ele pode ser executado nos sistemas operacionais GNU/Linux e MS Windows e é totalmente compatível com ACID e possui os seguintes recursos:

a) controle de concorrência multiversionado (MVCC);

b) recuperação em um ponto no tempo (PITR);

c) áreas de armazenamento (*Tablespaces*);

d) replicação assíncrona;

e) transações agrupadas (*Savepoints*);

f) cópias de segurança a quente (*online/hot backup*);

g) planejador de consultas sofisticado;

h) registrador de transações sequencial (WAL) para tolerância a falhas;

i) suporta conjuntos de caracteres internacionais.

Milani (2008), define PostgreSQL como um Sistema de Gerência de Banco de Dados estável e confiável, que disponibiliza os principais recursos dos sistemas relacionais pagos do mercado, apresentando as seguintes características:

a) alta disponibilidade: podendo atuar como um *cluster* de informações;

b) suporte a transações: suporte a operações ACID;

c) segurança e criptografia: suporte nativo a SSL;

d) *multithreads*: possibilita mais de uma conexão com o banco de dados, por meio de recurso de *multithreads;*

e) SQL: adota os padrões ANSI SQL;

f) capacidade de armazenamento: não possui limite máximo para um banco de dados;

g) incorporável em aplicações gratuitamente: possui licença BSD.

4.3.3.1 Limitações do PostgreSQL

De acordo com o manual do PostgreSQL (2015), ele é altamente escalável quanto a quantidade de dados e quanto ao número de usuários concorrentes. Segundo o autor, existem ambientes de produção que gerenciam mais de 5 *terabytes* (10^{12} *bytes*) de dados. No entanto, algumas limitações devem ser observadas, conforme Tabela 7.

Tabela 7 – Limitações do PostgreSQL

Limite	Valor
Tamanho máximo do banco de dados	Ilimitado
Tamanho máximo de uma tabela	32 TB
Tamanho máximo de uma linha de tabela	1.6 TB
Tamanho máximo de um registro	1 GB
Quantidade de linhas por tabela	Ilimitado
Quantidade de colunas por tabela	250 a 1600. Depende do tipo de coluna
Quantidade de index por tabela	Ilimitado

Fonte: Do autor, adaptado de PostgreSQL (2015).

4.3.4 MongoDB

ReadMond e Wilson (2012) definem MongoDB como um Sistema Gerenciador de Banco de Dados orientado a documentos, lançado em 2009 como um banco de dados escalável e flexível. O MongoDB armazena e processa documentos do tipo JSON e atualmente é utilizado em projetos como Foursquare[1], bit.ly[2] e no armazenamento de dados gerados na Organização Europeia para a Pesquisa Nuclear (CERN).

O funcionamento do MongoDB pode ser resumido na substituição de “linhas” do conceito relacional por um modelo mais flexível em “documentos”. O modelo é livre de esquema, de forma que atributos de um documento não sejam pré-definidos.

O sistema foi projetado desde o início para ser escalar, pois o modelo de armazenamento de documentos permite que a carga seja distribuída em mais servidores. Ele é capaz de balancear a carga automaticamente entre os servidores do *cluster*, de forma que os desenvolvedores não precisem se preocupar com a capacidade do servidor. Quando for

[1]https://pt.foursquare.com/
[2]https://bitly.com/

necessário aumentar a capacidade, é necessário apenas adicionar mais um servidor ao *cluster*. Além disso, quando um servidor do *cluster* parar de funcionar, automaticamente um nó escravo é promovido a um nó mestre, sem a necessidade de interferência de um administrador (CHODOROW; DIROLF, 2010).

Segundo Elco, Peter e Tim (2010, apud ROSA, 2013), o MongoDB utiliza JSON binário (BSON) ao invés de JSON para armazenar dados. Segundo o autor, o uso do BSON não altera a forma de como o usuário vai trabalhar com os dados, e ainda torna mais fácil a pesquisa e processamento de documentos pelo computador. Em contrapartida, o formato BSON ocupa mais espaço em disco.

O MongoDB não possui gerenciamento de transações e controle de concorrência, portanto o controle de modificações deve ser realizado em nível de aplicação, pois pode ocorrer de um usuário efetuar a escrita em um documento entre o processo de leitura e escrita de outro processo (TIWARI, 2010).

Segundo o manual do MongoDB (2016), o motor de armazenamento de dados utilizado por padrão até a versão 3.0 é o *MMAPv1 Storage Engine,* que tem como característica armazenar os dados em memória virtual e sincronizar os dados com o disco rígido em um intervalo de 60 segundos. Todas modificações feitas no banco são gravadas em disco rígido, para garantir que todas modificações sejam aplicadas em caso de uma pane. O protótipo desenvolvido utiliza essa estratégia de armazenamento. De acordo com o autor, os dados sincronizados para o disco rígido são armazenados de forma contígua, em arquivos com espaço pré-alocado. Quando uma base de dados é criada, ela possui 64MB alocados em disco, quando metade deste espaço é ocupado, o banco aloca mais 128MB, esse processo continua alocando mais 256MB, 512MB, 1024MB, até atingir 2048MB, após isso, novas alocações serão de 2048MB.

De acordo com Lóscio, Oliveira e Pontes (2011), o MongoDB pode ser utilizado em diferentes sistemas operacionais e em diversas linguagens como: C, C#, C++, Java, Perl, Python, Ruby e PHP. Segundo os autores, o modelo de dados do MongoDB é composto por:

a) uma coleção é armazenada em um banco de dados;

b) um documento é armazenado em uma coleção;

c) um documento pode ser entendido como um conjunto de campos;

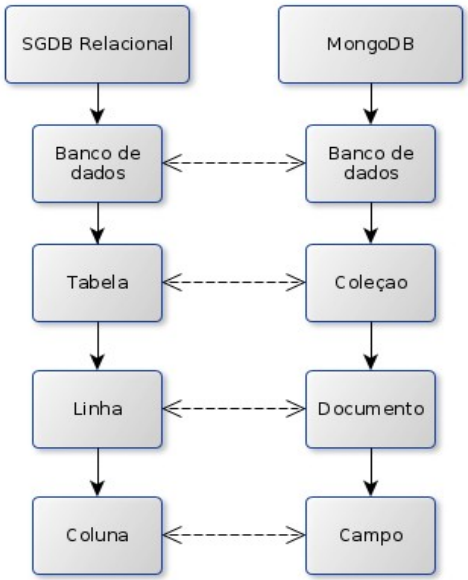
d) um campo pode ser entendido como um par chave-valor;

e) uma chave é um atributo, composto por um nome;

f) um valor pode ser um carácter, inteiro, ponto flutuante, um documento, um vetor.

Na Figura 13, é apresentada a comparação do modelo de dados presente no MongoDB com o modelo presente em bancos de dados relacionais.

Figura 13 – Equivalência entre SGDB relacional e MongoDB



Fonte: Elaborado pelo autor (2015).

Nas próximas subseções serão apresentadas as principais características dos bancos de dados criados no sistema MongoDB.

4.3.4.1 Indexação no MongoDB

De acordo com o manual do MongoDB (2015), o SGDB fornece diferentes tipos de índices, que podem ser criados em qualquer campo ou documento. Assim como nos bancos relacionais, o objetivo dos índices é diminuir a quantidade de documentos processados. Segundo o autor, o Mongo suporta os seguintes tipos de índices:

a) índices em único campo: é possível criar índice para um único campo do documento ou subdocumentos em uma coleção;

b) índices em campos compostos: esse índice permite indexar mais de um campo de documentos em uma coleção;

c) índices multi-chave: é um índice para campos do tipo matriz, é utilizado uma chave de índice para cada valor pertencente a matriz;

d) índices geoespaciais: índices que suportam pesquisas baseadas em localização em dados armazenados em documentos GeoJSON ou coordenadas;

e) índice de texto: índices que atendem pesquisa em campos texto de documentos;

f) índice *hash*: índices utilizados para consultas utilizando operador de igualdade.

4.3.4.2 Agregação

Segundo o manual MongoDB (2015), o sistema oferece um vasto conjunto de operações de agregação que podem executar cálculos ou examinar um conjunto de dados. Operações de agregação utilizam coleções de dados como entrada e retornam o resultado em forma de um ou mais documentos.

De acordo com o autor, ele permite os seguintes tipos de agregação:

a) *Pipeline*: utiliza o conceito de *pipeline* de processamento de dados, onde os dados são processados em múltiplos estágios em sequência e o resultado de uma etapa é utilizado na entrada de outra. Algumas etapas fornecem filtros que podem ser utilizados para consultar ou transformar documentos e outras fornecem ferramentas para o agrupamento e classificação de documentos, podendo agregar também conteúdo de matrizes. Essas ferramentas são capazes de calcular a média, somar números ou concatenar uma *String*;

b) *Map/Reduce*: o MongoDB utiliza as funções Javascript personalizadas para executar agregação em *Map/Reduce*, além das fases de mapeamento e redução, ele disponibiliza um estágio final que pode fazer modificações no resultado, assim como agregação por *Pipeline*, ele especifica uma condição de consulta aos documentos de entrada classificando e limitando o resultado;

c) operações únicas: o MongoDB oferece operações individuais de agregação, essas operações podem ser entendidas como comandos especiais que possuem um único propósito. Operações comuns são: retornar valores distintos de um campo, agrupar valores e obter a contagem de documentos que correspondam a uma determinada condição. Essas operações não possuem a mesma capacidade e flexibilidade do *Pipeline* e *Map/Reduce*.

4.3.5 Outras tecnologias

Além dos bancos de dados PostgreSQL e MongoDB, outras tecnologias foram utilizadas na elaboração de um protótipo capaz de efetuar testes de stress, volume e carga:

a) Apache HTTP Server[3]: é um servidor web de código aberto. O Apache tem como vantagem suporte ao HTTP 1.1, suporte a SSL, suporte a CGI, Perl e PHP, possui *logs* customizáveis e configuração simples. Outra vantagem é o fato de ser gratuito e estar sob licença GNU Public Licence (THE APACHE SOFTWARE FOUNDATION, 2015);

b) PHP[4]: é uma linguagem script *open source* de uso geral cujo nome é um acrônimo recursivo para PHP: Hypertext Preprocessor, ela surgiu em 1994, foi criada por RasmusLerdof e foi escrita através da linguagem de programação C. É adequada para o desenvolvimento web (PHP, 2015);

c) HTML[5]: é uma linguagem utilizada para desenvolver as páginas na internet (HTML, 2015);

d) JavaScript[6]: é uma linguagem multi-paradigma, leve, orientado a objetos e interpretada pelo próprio navegador (JAVASCRIPT, 2015);

e) CSS[7]: é uma linguagem de folhas de estilo utilizada em páginas da web. Seu objetivo é prover um formato para o documento HTML (W3SHOOLS, 2015);

[3]http://httpd.apache.org/
[4]http://www.php.net
[5]http://www.w3.org/html/
[6]https://developer.mozilla.org/en-US/docs/Web/JavaScript
[7]http://www.w3schools.com/css/css_intro.asp

f) Hightcharts.js[8]: é uma biblioteca em JavaScript puro para criação de gráficos interativos para aplicações web (HIGHCHARTS, 2016);

g) Laravel[9]: é um *framework* PHP criado por Taylor Otwell que utiliza o padrão MVC. Possui gerenciador de dependências, sintaxe simples e utilitários que auxiliam o desenvolvimento de sistemas (LARAVEL, 2016).

4.4 Levantamento de requisitos

Para que este trabalho pudesse atingir os objetivos estabelecidos, o protótipo desenvolvido precisou atender determinados requisitos, que serão apresentados na subseções seguintes, e estarão divididos entre funcionais e não funcionais.

4.4.1 Requisitos funcionais

A seguir, serão apresentados os requisitos funcionais que são atendidos pelo protótipo:

a) cadastrar *tags*: através desse requisito o usuário é capaz de cadastrar metadados MARC que possuem o objetivo de descrever os materiais catalogados na base de dados;

b) cadastrar tipo: através desse requisito o usuário é capaz de manter tipos para os materiais e seus exemplares (livro, monografia, tese, dissertação, norma);

c) cadastrar tipo físico: através desse requisito o usuário é capaz de manter tipos físico para os materiais e seus exemplares (impresso, cd, dvd);

d) cadastrar materiais: através desse requisito o usuário é capaz de cadastrar e editar materiais. O cadastro apresenta uma estrutura onde é permitido cadastrar qualquer campo MARC;

e) pesquisar materiais: através desse requisito o usuário é capaz de recuperar materiais catalogados no banco de dados. É possível utilizar qualquer campo

[8]http://www.highcharts.com/products/highcharts
[9]https://laravel.com/

MARC como filtro. Na exibição de resultados, ao lado do registro, é possível excluir ou editar o registro;

f) importar materiais: através desse requisito o usuário é capaz de importar materiais armazenados em ISO2709;

g) mostrar tempo de execução ao inserir, excluir e editar registros para ambos os bancos de dados;

h) executar teste de busca: através desse requisito o usuário é capaz de executar testes de buscas em ambos os SGDB's;

i) executar teste de tolerância a falhas: através desse requisito o usuário é capaz de executar um ou mais testes de tolerância a falhas com objetivo de verificar o estado do banco de dados após a ocorrência de uma falha simulada;

j) executar teste de carga: através desse requisito o usuário é capaz de executar um ou mais testes de carga programados em ambos bancos de dados, com objetivo de verificar a performance, quando submetido a um grande número de transações que inserem novos materiais;

k) executar teste de volume: através desse requisito o usuário é capaz de executar um ou mais testes de volume programados em ambos bancos de dados, com objetivo de verificar o seu comportamento ao lidar com um grande volume de dados;

l) executar teste de stress: através desse requisito o usuário é capaz de executar um ou mais testes de stress programados em ambos bancos de dados, com objetivo de avaliar o comportamento do banco de dados, conforme o número de acessos simultâneos em função do tempo de resposta;

m) exportar resultados para CSV: através desse requisito o usuário é capaz de exportar os resultados dos testes em formato CSV.

4.4.2 Requisitos não funcionais

A seguir, serão apresentados os requisitos não funcionais que são atendidos pelo protótipo:

a) desenvolvido na linguagem PHP: esse requisito define a linguagem que foi utilizada no desenvolvimento do protótipo. A linguagem foi escolhida por ser a linguagem utilizada pelo *software* Gnuteca, além de ser prática, *open-source* e suportar ambos bancos de dados analisados neste trabalho;

b) utilizar funções nativas do PHP para SGDB PostgreSQL como *pg_connection*, *pg_query*, *pg_exec* e *pg_fetch_array*;

c) utilizar a extensão oficial Mongo driver do PHP para o SGDB MongoDB.

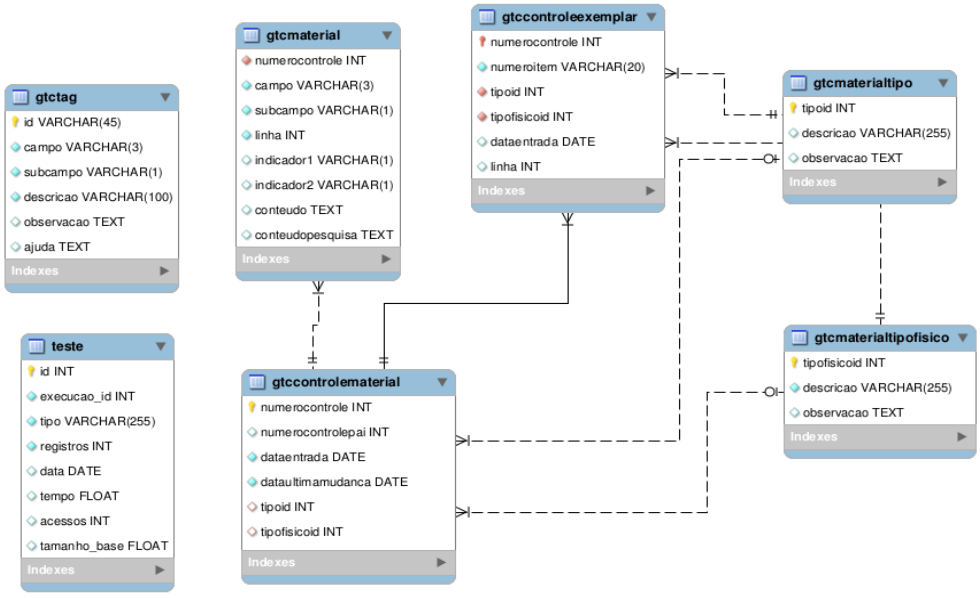
4.5 Modelagem da base de dados

O modelo lógico utilizado para o desenvolvimento do protótipo é baseado no modelo utilizado pelo *software* Gnuteca. O modelo original não foi utilizado devido ao seu tamanho e complexidade. Portanto, foram identificadas e simplificadas as principais entidades necessárias para o armazenamento de registros bibliográficos. Esse modelo permite que diferentes tipos de objetos possam ser catalogados na base de dados.

Conforme Figura 14, o conjunto de campos e subcampos disponíveis para serem utilizados estão armazenados na entidade “gtctag”. Os registros MARC são armazenados na entidade “gtcmaterial”, portanto, um registro bibliográfico possui múltiplos registros nessa entidade. Com o objetivo de auxiliar no controle de obras e exemplares, foram utilizadas as entidades “gtccontroleexemplar”, que armazena os exemplares e a tabela “gtccontrolematerial” para controle da obra.

Além das entidades citadas anteriormente, o modelo apresenta as entidades auxiliares “gtcmaterialtipo” e “gtcmaterialtipofisico”, para armazenar o tipo e o tipo físico respectivamente e a entidade “teste” para armazenar os resultados da aplicação dos testes de carga, volume e stress no PostgreSQL.

Figura 14 – Diagrama ER para PostgreSQL



Fonte: Elaborado pelo autor (2015).

O modelo NoSQL não exige que seja criada uma estrutura prévia para armazenamento dos dados, devido ao fato de ser livre de esquema. Portanto, para o armazenamento dos dados em MongoDB foram utilizadas coleções para armazenar tipos, tipos físicos, *tags* e materiais.

Na Figura 15 é apresentado um exemplo de documento que contém um registro bibliográfico, contido na coleção “material”. O documento apresenta as mesmas informações contidas na entidade “gtccontrolematerial” do modelo relacional. As etiquetas que estão armazenadas na entidade “gtcmaterial” estão contidas no vetor “etiquetas”, e os exemplares que estão contidos na entidade “gtccontroleexemplar” estão armazenados no vetor “exemplar”.

Figura 15 – Coleção de documentos onde estão armazenados os metadados do objeto/material

```
1
2   {
3       "_id" : ObjectId("56ef44ea70b7b5ba148b58f0"),
4       "numerocontrole" : "2",
5       "dataentrada" : "20/03/2016",
6       "tipoid" : {
7           "$ref" : "tipo",
8           "$id" : ObjectId("56ed5b2870b7b5980c8b4567")
9       },
10
```

Continua

```
    "tipofisicoid" : {
        "$ref" : "tipofisico",
        "$id" : ObjectId("56ed5c0570b7b5980c8b4568")
    },
    "etiquetas" : [ {
            "numerocontrole" : "2",
            "campo" : "245",
            "subcampo" : "a",
            "linha" : "0",
            "indicador1" : null,
            "indicador2" : null,
            "conteudo" : "Teste de carga 1",
            "conteudopesquisa" : "TESTE DE CARGA 1"
        },
    ],
    "exemplar" : [ {
            "numerocontrole" : "2",
            "numeroitem" : "1",
            "tipoid" : {
                "$ref" : "tipo",
                "$id" : ObjectId("56ed5b2870b7b5980c8b4567")
            },
            "tipofisicoid" : {
                "$ref" : "tipofisico",
                "$id" : ObjectId("56ed5c0570b7b5980c8b4568")
            },
            "dataentrada" : "20/03/2016",
            "linha" : "0"
        },
    ]
}
```

Fonte: Elaborado pelo autor (2016). Conclusão

As *tags* que são armazenadas na entidade “gtctag” do modelo relacional são armazenadas na coleção “tag” no MongoDB, conforme Figura 16.

Figura 16 – Coleção de documentos onde serão são armazenados os metadados disponíveis

```

{
    "_id" : ObjectId("56e74ef070b7b5d2108b4567"),
    "campo" : "100",
    "subcampo" : "a",
    "descricao" : "Autor",
    "observacao" : "Autor",
    "ajuda" : "te",
    "id" : "637291",
}
```

Fonte: Elaborado pelo autor (2016).

A Figura 17 contém um exemplo de um documento de tipo de material, contido na coleção “tipo”. Esse documento armazena os mesmos registros contidos na entidade “gtcmaterialtipo” no modelo relacional.

Figura 17 – Coleção de documentos onde são armazenados os tipos de materiais

```
1
2  {
3       "_id" : ObjectId("56ed5b2870b7b5980c8b4567"),
4       "descricao" : "Livro",
5       "observacao" : "",
6       "id" : "19"
7  }
```

Fonte: Elaborado pelo autor (2016).

Os registros de tipo físico de material que são armazenados na entidade "gtcmaterialtipo" no modelo relacional são armazenados na coleção "tipo", conforme Figura 18.

Figura 18 – Coleção de documentos onde são armazenados os tipos físico de materiais

```
1
2  {
3       "_id" : ObjectId("56ed5c0570b7b5980c8b4568"),
4       "descricao" : "Impresso",
5       "observacao" : "",
6       "id" : "13"
7  }
```

Fonte: Elaborado pelo autor (2016).

Os resultados de testes de carga, volume e stress aplicados no MongoDB são armazenados na coleção *teste*, conforme Figura 19.

Figura 19 – Coleção de documentos onde são armazenados os resultados dos testes

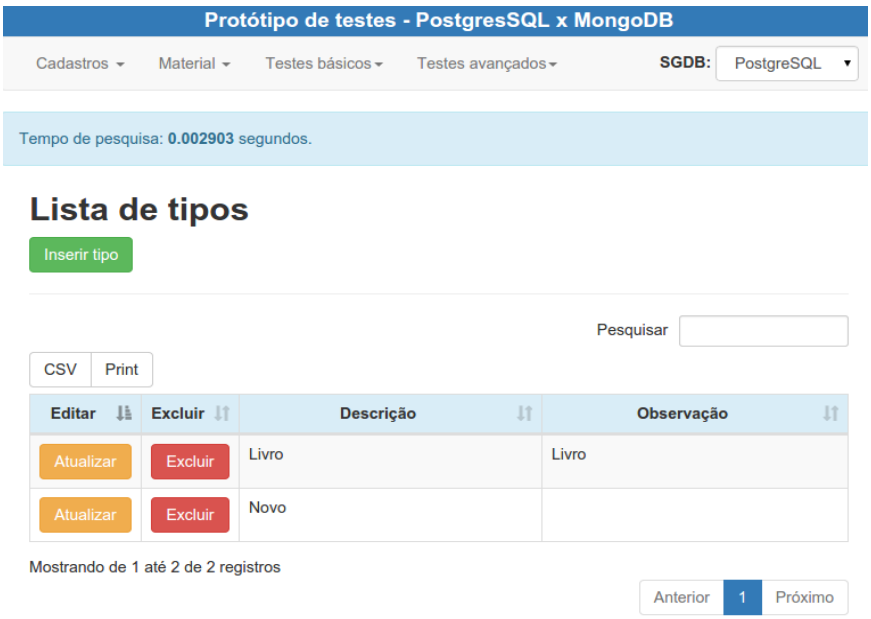
```
1
2  {
3       "_id" : ObjectId("56ef46dd70b7b5a00e8b457a"),
4       "execucaoid" : "1",
5       "tipo" : "stress",
6       "registros" : "105000",
7       "data" : "21/03/2016",
8       "tempo" : "0.022485",
9       "acessos" : "10",
10      "tamanhobase" : null
11 }
```

Fonte: Elaborado pelo autor (2016).

4.6 Protótipo

Nesta seção é apresentado um panorama geral do protótipo desenvolvido. Conforme Figura 20, as etiquetas (*tags*), tipos e tipos físicos podem ser mantidos através do acesso no menu “Cadastros”. O tempo das operações (listar, obter registro, editar, excluir e inserir) são medidos e apresentados na parte superior da interface, abaixo do menu. As operações inserir, excluir e editar são aplicadss em ambos SGDB, já a listagem é obtida do SGDB que estiver selecionado no canto superior direito.

Figura 20 – Interface para cadastro de tipos

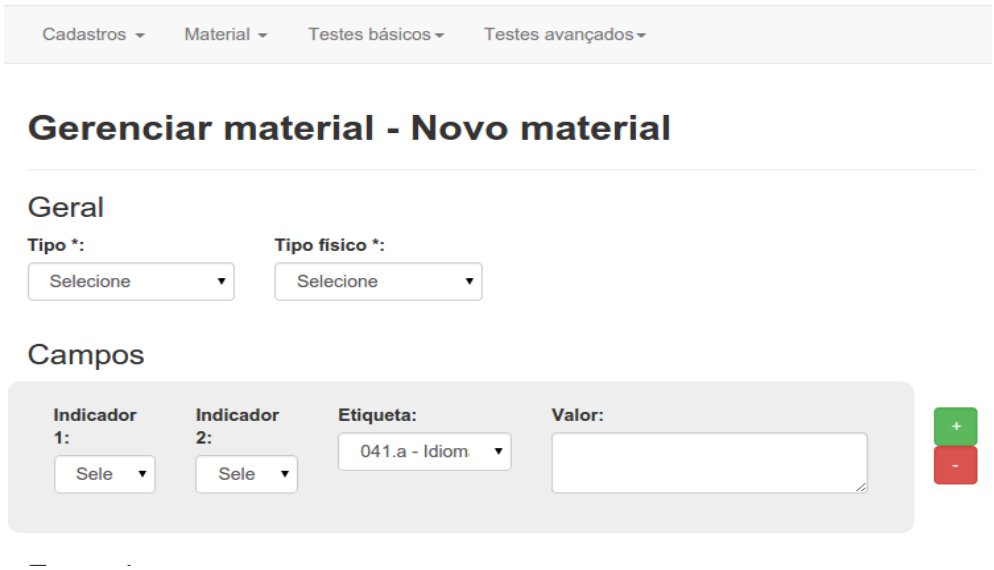


Fonte: Elaborado pelo autor (2016).

Os registros de materiais são mantidos no menu “Material”, onde o usuário é capaz de inserir, pesquisar e importar materiais. Para inserir um novo material, o usuário deve informar o tipo, tipo físico, campos MARC e exemplares. A importação de material atende o cadastro de materiais que estão armazenados no formato ISO2709. Para realizar a importação o usuário deve informar o tipo e tipo físico, também deve informar o arquivo ISO e os separadores de registro, campo e subcampo.

A Figura 21 apresenta a interface de cadastro de materiais, onde o usuário deve informar o tipo e o tipo físico do material, preencher as etiquetas como: título, autor, editora, local de publicação, e informar os exemplares.

Figura 21 – Interface para cadastro de materiais



Fonte: Elaborado pelo autor (2016).

Conforme Figura 22, a pesquisa de material permite que a busca seja efetuada tanto no PostgreSQL quanto no MongoDB (o SGDB pode ser alterado na parte superior). A interface permite a aplicação de múltiplos filtros através de combinações de etiquetas e operadores booleanos.

Figura 22 – Interface de pesquisa de materiais

Fonte: Elaborado pelo autor (2016).

A aplicação dos testes de desempenho são realizadas na opção “Novo teste completo” no menu “Testes avançados”, a Figura 23 apresenta essa interface, onde o usuário deve informar o número da execução (que será utilizada na análise), os testes que serão executados e a quantidade de materiais.

Figura 23 – Novo teste completo de desempenho

Fonte: Elaborado pelo autor (2016).

O teste de tolerância a falhas é executado na opção “Teste de tolerância a falhas” no menu “Testes básicos”. A interface pode ser vista na Figura 24, e nela o usuário deve informar a quantidade de materiais que serão inseridas e o ponto em que ocorrerá uma falha.

Figura 24 – Interface do teste de tolerância a falhas

Fonte: Elaborado pelo autor (2016).

O resultado dos testes de desempenho são visualizados no menu “Testes avançados” (carga, volume e stress). As interfaces de teste permitem que um novo teste seja executado, que os resultados individuais sejam visualizados e também a visualização do teste consolidado (FIGURA 25).

Figura 25 – Interface de visualização de resultados de testes

Fonte: Elaborado pelo autor (2016).

4.7 Metodologia de teste

Nesta seção é apresentado a metodologia dos testes que foram aplicados nos SGDB's PostgreSQL e MongoDB. Foram aplicados testes de busca, carga, volume, stress e tolerância a falhas.

4.7.1 Teste de busca

O teste de busca tem o objetivo de determinar o desempenho dos SGDB's em lidar com busca de materiais. O teste consiste em medir o tempo do banco de dados ao buscar por materiais cadastrados, combinando campos MARC nos filtros. O teste foi realizado de duas maneiras, devido às particularidades da linguagem utilizada quanto ao retorno de registros. Uma estratégia foi analisar o desempenho de consultas que retornam dados e a segunda maneira é a execução de consultas que apenas contam registros. O principal objetivo é

analisar o desempenho da busca, isolando a obtenção de dados. Na seção de resultados as particularidades serão exploradas e os resultados comparados.

4.7.2 Teste de carga

O teste de carga tem o objetivo de determinar a aceitabilidade do banco de dados em lidar com cargas de trabalhos variáveis em função do tempo. O teste consiste na inserção de múltiplos materiais em ambos SGDB's. O usuário define a quantidade de materiais inseridos em cada execução. Para cada material inserido são necessários 21 registros no PostgreSQL (um registro na tabela "gtccontrolematerial", dez registros na tabela "gtcmaterial" e dez registros na tabela "gtccontroleexemplar"). Apesar do MongoDB necessitar de apenas um documento para armazenar um material, o tempo de execução estará associado a quantidade de registros necessários para o PostgreSQL, pois ambos SGDB's conterão a mesma informação. O algoritmo do teste insere apenas cinco materiais diferentes, pois precisou garantir que em ambos algoritmos sempre gravassem os mesmos dados, quando selecionada determinada quantidade de materiais.

4.7.3 Teste de volume

O teste de volume tem o objetivo de verificar a capacidade que o SGDB possui em lidar com grande volume de dados em relação ao tempo. Nesse teste são executadas consultas que retornam todos os materiais armazenados e não contempla a extração dos dados, ou seja, mede o tempo da execução da consulta e não da obtenção dos dados, a partir dos recursos ou cursores retornados em cada caso.

4.7.4 Teste de Stress

O teste de stress tem o objetivo de avaliar o comportamento do SGDB quando submetido em condições extremas. Nesse teste o algoritmo de teste de volume é executado de forma simultânea em uma determinada massa de dados, e o tempo de resposta é medido.

4.7.5 Teste de tolerância a falhas

O teste de tolerância a falhas tem o objetivo de avaliar o comportamento do SGDB quando submetido à uma falha simulada em uma operação de inserção de dados, para a análise, o teste apaga todos os materiais antes de realizar inserção. Nesse teste o usuário informa a quantidade de materiais que devem ser inseridos e também informa o ponto em que ocorrerá a falha (após qual material a falha ocorrerá). Embora os testes anteriores meçam as operações em relação ao tempo, esse teste avalia o estado do banco de dados após uma falha.

4.8 Método estatístico

Os testes de desempenho de carga, volume e stress são executados 10 vezes completamente, onde uma execução completa é composta pela repetição do teste para 1, 10, 100, 1.000, 5.000, 10.000, 25.000, 50.000 materiais. Para cada material são considerados 21 registros, portanto, os testes são aplicados sobre 21, 210, 2.100, 105.000, 210.000, 525.000 e 1.050.000. Conforme já mencionado, um documento de material no MongoDB equivale a 21 registros no PostgreSQL, portanto os resultados serão agrupados pela equivalência, ou seja, o resultado da aplicação de 21 registros no modelo relacional será agrupado com a aplicação em um documento no MongoDB. Para a apresentação consolidada dos testes, o melhor e o pior tempo são descartados, resultando em 8 aplicações completas. Sobre elas é realizada uma média aritmética simples para o tempo na maioria dos testes e para o espaço em disco no teste de carga.

4.9 Ambiente de teste

Esta seção tem o objetivo de apresentar os componentes de *hardware* e *sofware* envolvidos, versões dos SGDB's e as configurações dos mesmos.

4.9.1 Componentes de *hardware* e *software*

Para a aplicação dos testes, foi utilizado um *notebook* com sistema operacional Ubuntu Desktop 14.04 LTS, composto pelos seguintes componentes de *hardware*: processador Intel(R) Core i7-4500 1.80GHz, memória RAM de 8GB DDR3 1600MHz e unidade de armazenamento SSD com capacidade de 240GB.

4.9.2 Versões e configurações dos SGDB's

Os testes foram aplicados nos bancos de dados PostgreSQL 9.3.6 e MongoDB 3.0.2. De acordo com Boaglio (2015), o MongoDB não necessita de ajustes de configurações de performance como nos outros SGDB's, pois ele assume essa tarefa consumindo os recursos de acordo com a necessidade do banco. Ainda segundo o autor é comum que o MongoDB aloque uma coleção inteira na memória RAM para manter a performance.

Portanto, para a aplicação dos testes no PostgreSQL foi necessário realizar um ajuste nos parâmetros de performance, para se igualar ao MongoDB. A Figura 26 apresenta os parâmetros de performance que foram modificados.

Figura 26 – Configurações utilizadas no PostgreSQL

```
1
2  default_statistics_target = 50 # pgtune wizard 2016-04-03
3  maintenance_work_mem = 480MB # pgtune wizard 2016-04-03
4  constraint_exclusion = on # pgtune wizard 2016-04-03
5  checkpoint_completion_target = 0.9 # pgtune wizard 2016-04-03
6  effective_cache_size = 5632MB # pgtune wizard 2016-04-03
7  work_mem = 48MB # pgtune wizard 2016-04-03
8  wal_buffers = 8MB # pgtune wizard 2016-04-03
9  checkpoint_segments = 16 # pgtune wizard 2016-04-03
10 shared_buffers = 1920MB # pgtune wizard 2016-04-03
11 max_connections = 80 # pgtune wizard 2016-04-03
```

Fonte: Elaborado pelo autor (2016).

5 RESULTADOS E DISCUSSÃO

Este capítulo tem o objetivo de apresentar os resultados obtidos nos testes de desempenho de busca, carga, volume, stress e tolerância a falhas executados através do protótipo desenvolvido e uma avaliação qualitativa da implementação em relação à impedância dos SGDB's avaliados. Os resultados apresentam trechos de código fonte, que também podem ser consultados no Apêndice B deste trabalho.

5.1 Teste de busca

O teste de de busca visa medir o tempo dos SGDB's ao buscar materiais. A Figura 27 apresenta um exemplo de uma busca com filtros combinados no PostgreSQL onde o título (245.a) é "BANCOS DE DADOS" e o autor (100.a) é "DATE".

Na consulta foi necessário realizar uma série de junções de tabelas, tanto para poder aplicar o filtro, quanto para obter as informações de autor, título, subtítulo, tipo e tipo físico.

Figura 27 – Busca de materiais no PostgreSQL

```
1
2   SELECT controle.numerocontrole,
3              titulo.conteudo as titulo,
4              subtitulo.conteudo as subtitulo,
5              autor.conteudo as autor,
6              tipo.descricao as tipo,
7              tipofisico.descricao as tipofisico
8         FROM gtccontrolematerial controle
9   LEFT JOIN gtcmaterial titulo
10         ON (titulo.numerocontrole = controle.numerocontrole AND titulo.campo =
    '245' AND titulo.subcampo = 'a' AND titulo.linha = 0)
```

Continua

```
11 LEFT JOIN gtcmaterial subtitulo
12       ON (subtitulo.numerocontrole = controle.numerocontrole AND subtitulo.campo
   = '245' AND subtitulo.subcampo = 'b' AND subtitulo.linha = 0)
13 LEFT JOIN gtcmaterial autor
14      ON (autor.numerocontrole = controle.numerocontrole AND autor.campo = '100'
   AND autor.subcampo = 'a' AND autor.linha = 0)
15 LEFT JOIN gtcmaterialtipo tipo
16        ON (controle.tipoid = tipo.id)
17 LEFT JOIN gtcmaterialtipofisico tipofisico
18        ON (controle.tipofisicoid = tipofisico.id)
19     WHERE controle.numerocontrole IN (
20 SELECT  distinct(numerocontrole)  as  numerocontrole  FROM  gtcmaterial  WHERE
   ((   (campo = '245' AND subcampo = 'a' AND conteudopesquisa ILIKE '%BANCOS DE
   DADOS%')) OR (campo = '100' AND subcampo = 'a' AND conteudopesquisa ILIKE '%DATE
   %')))
```

Fonte: Elaborado pelo autor (2016). Conclusão

A Figura 28 apresenta o mesmo exemplo anterior executado no MongoDB.

Figura 28 – Busca de materiais no MongoDB

```
1
2  db.material.find( {$or : [   { etiquetas: { $elemMatch: { 'campo': '245',
   'subcampo' : 'a', "conteudopesquisa" : /BANCO DE DADOS/ } } }, { etiquetas:
   {  $elemMatch: {'campo': '100', 'subcampo' : 'a', 'conteudopesquisa': /DATE/}  }
   } ]})
```

Fonte: Elaborado pelo autor (2016).

Conforme Figura 27, a consulta executada obtém a descrição do tipo e tipo físico através de junções para cada material recuperado, portanto, para obter o mesmo resultado no MongoDB é necessário resolver as referências via aplicação. Para isso, é necessário iterar sobre o resultado e resolver a referência através das consultas apresentadas na Figura 29, onde "56fc69a270b7b58b058b4568" e "56fc6a7e70b7b587058b456e" são referências para tipo e tipo físico, respectivamente. Para otimizar a consulta ao banco, as referências resolvidas são guardadas em um vetor e utilizadas quando necessário, para que referências iguais não sejam executadas novamente no banco.

Figura 29 – Busca de referências para tipo e tipo físico

```
1
2  db.tipo.find({ _id : ObjectId("56fc69a270b7b58b058b4568")}, {descricao:true})
3  db.tipofisico.find({_id : ObjectId("56fc6a7e70b7b587058b456e")},
   {descricao:true})
```

Fonte: Elaborado pelo autor (2016).

O PostgreSQL e o MongoDB retornam resultados diferentes ao realizar uma consulta. O primeiro carrega os dados resultantes em memória, o segundo retorna um cursor. Essa diferença foi percebida na implementação do protótipo.

Na implementação das buscas de dados em PostgreSQL, a função nativa *pg_query* utilizada para obter dados do PostgreSQL retorna um *resource*, e, segundo PHP (2016), funções como *pg_fetch_array* ou *pg_fetch_object* devem ser utilizadas para obter os dados do *resource*. Já para a implementação das buscas em MongoDB, a função *find* retorna um cursor, mais precisamente um objeto *MongoCursor.* De acordo com o autor, esse objeto cursor mantém uma conexão aberta com o banco de dados, e funções como *iterator_to_array* ou *foreach* devem ser utilizadas para extrair os dados. Portanto, esse teste é executado de duas formas, a primeira delas é pela interface de busca do protótipo, onde os dados são apresentados em uma listagem, nesse caso é avaliado as performances dos SGBD's em buscar e extrair os dados.

Figura 30 – Exemplo de contagem de registros no PostgreSQL

```
1
2   SELECT count(*)
3        FROM gtccontrolematerial controle
4   LEFT JOIN gtcmaterial titulo
5         ON (titulo.numerocontrole = controle.numerocontrole AND titulo.campo =
    '245' AND titulo.subcampo = 'a' AND titulo.linha = 0)
6   LEFT JOIN gtcmaterial subtitulo
7         ON (subtitulo.numerocontrole = controle.numerocontrole AND subtitulo.campo
    = '245' AND subtitulo.subcampo = 'b' AND subtitulo.linha = 0)
8   LEFT JOIN gtcmaterial autor
9        ON (autor.numerocontrole = controle.numerocontrole AND autor.campo = '100'
    AND autor.subcampo = 'a' AND autor.linha = 0)
10       WHERE controle.numerocontrole IN (
11  SELECT distinct(numerocontrole) as numerocontrole FROM gtcmaterial WHERE
    ((  (campo = '245' AND subcampo = 'a' AND conteudopesquisa ILIKE '%BANCOS DE
    DADOS%')) OR (campo = '100' AND subcampo = 'a' AND conteudopesquisa ILIKE '%DATE
    %')))
```

Fonte: Elaborado pelo autor (2016).

A segunda maneira é realizar uma contagem de registros ao invés de uma seleção, o resultado será apenas um número, portanto, o tempo utilizado para extrair os resultados da consulta não é um fator relevante. A Figura 30 apresenta um exemplo semelhante à Figura 31, onde é realizada uma contagem de registros ao invés de uma projeção de campos em uma consulta PostgreSQL. A Figura 28 apresenta o equivalente para o MongoDB.

Figura 31 – Exemplo de contagem de registros no MongoDB

```
1
2  db.material.find( {$or : [   { etiquetas: { $elemMatch: { 'campo': '245',
   'subcampo' : 'a', "conteudopesquisa" : /BANCO DE DADOS/ } } }, { etiquetas:
   {  $elemMatch: {'campo': '100', 'subcampo' : 'a', 'conteudopesquisa': /DATE/}  }
   } ]}).count()
```

Fonte: Elaborado pelo autor (2016).

Com objetivo de otimizar as consultas, foram criados índices em ambos SGDB's. No PostgreSQL foi criado um índice na tabela “gtcmaterial” para os campos “campo”, “subcampo” e “conteudopesquisa”. A criação do índice é apresentada na Figura 32.

Figura 32 – Criação do índice *test_index* para PostgreSQL

```
1
2  CREATE INDEX test_index ON gtcmaterial (campo, subcampo, conteudopesquisa
   varchar_pattern_ops);
```

Fonte: Elaborado pelo autor (2016).

Conforme apresentado no subseção 4.3.4.1, o MongoDB também permite a criação de índices, portanto foi criado um índice na coleção “material” para os campos “etiquetas.campo”, “etiquetas.subcampo” e “etiquetas.conteudopesquisa”. A criação do índice é apresentada na Figura 33.

Figura 33 – Criação do índice para MongoDB

```
1
2  db.material.createIndex({"etiquetas.campo":1,"etiquetas.subcampo":1,
   "etiquetas.conteudopesquisa":1})
```

Fonte: Elaborado pelo autor (2016).

A execução dos testes foi realizada através de duas maneiras, na primeira delas foi utilizar a interface de busca de materiais do protótipo localizada em Menu → Material → Pesquisar. Essa interface obtém e exibe os resultados obtidos na consulta. A segunda maneira foi utilizar a interface de teste de busca localizada em Menu → Testes básicos → Teste de busca. Essa interface exibe a quantidade de registros e tempo de execução da consulta em ambos SGDB's. Ambas as interfaces permitem a combinação de filtros através de seleção de

campos e operadores booleanos (e/ou). Para a obtenção de resultados, foi utilizada a mesma combinação de filtros em ambas interfaces. A Tabela 8 apresenta os resultados.

Tabela 8 – Resultado do teste de busca

Condição	Quantidade (registros)	Postgres (interface)	Postgres (teste)	Mongo (interface)	Mongo (teste)
245.a CONTEM 'BANCO' E 650.a CONTEM 'INFORMATICA' E 100.a CONTEM 'el'	10000	0,489892s	0,446063s	1,541534s	0,232550s
245.a CONTEM 'BANCO' E 650.a CONTEM 'INFORMATICA' OU 100.a CONTEM 'el'	40000	1,001012s	0,831537s	5,965069s	0,458339s
245.a CONTEM 'SISTEMAS' E 260.a CONTEM 'SAO PAULO' E 650.a CONTEM 'banco'	20000	2,870094s	0,613667s	2,984691s	0,381753s
245.a CONTEM 'SISTEMAS' E 260.a CONTEM 'RIO' E 650.a CONTEM 'banco'	10000	0,490941s	0,451199s	1,734503s	0,386137s
245.a CONTEM 'SISTEMAS' E 260.a CONTEM 'RIO' OU 650.a CONTEM 'banco' E 100.a CONTEM 'Garcia'	10000	0,505619s	0,472283s	1,571384s	0,267236s

Fonte: Elaborado pelo autor (2016).

Os resultados apontam uma performance superior do MongoDB em relação ao PostgreSQL quando utilizada a interface do teste de busca, esse resultado demonstra a performance a nível de banco. Porém, o resultado pela interface de busca demonstra que o PostgreSQL possui performance superior a nível de aplicação. Isso ocorreu porque a interface de busca retorna todos os resultados, ou seja, ela retorna todos os dados do *resource* e do cursor. Iterar sobre dados do *resource* do PostgreSQL é mais rápido do que iterar sobre todo o cursor do MongoDB, a diferença entre eles aumenta proporcionalmente de acordo com a quantidade de registros. Isso acontece porque o primeiro deles obtém dados armazenados na memória RAM, e o segundo mantém uma conexão aberta com o banco de dados. O MongoDB foi concebido para grande volume de dados, na ordem de *terabytes* de informação, seria impossível nesse cenário o MongoDB retornar dados sem utilizar um cursor, pois não haveria memória disponível para isso. No PostgreSQL esse mesmo cenário teria de utilizar os recursos de *limit* e *offset* para não exceder o limite de memória. Uma alternativa para diminuir essa diferença seria adaptar o protótipo para mostrar os dados de 20 em 20 resultados, navegando no cursor do MongoDB e nos resultados do *resource* em memória.

5.2 Teste de carga

O teste de carga foi executado de duas formas, onde os materiais são inseridos um a um e em bloco. A Figura 34 contém o código do algoritmo 1 utilizado no PostgreSQL, onde cada registro é inserido em uma operação distinta (um a um). No algoritmo o valor "$registros" é informado pelo usuário.

Figura 34 – Algoritmo de carga 1 para PostgreSQL

```
1
2  $this->iniciar();
3  for ( $i = 0; $i < $registros; $i++ ) {
4    $numeroControle = ControleMaterial::proximoNumeroDeControlePg();
5    $sqlControle = $this->obterSQLControle($numeroControle);
6    $db->execute($sqlControle);
7    $sqlsMaterial = $this->obterSQLMaterial($numeroControle, $registros);
8
9    foreach ( $sqlsMaterial as $sql ) {
10     $db->execute($sql);
11   }
12
13   $sqlsExemplar = $this->obterSQLExemplar($numeroControle);
14
15   foreach ( $sqlsExemplar as $sql ) {
16     $db->execute($sql);
17   }
18 }
19
20 $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

O código do algoritmo 1 para o MongoDB, onde os documentos são inseridos um a um, é representado pela Figura 35.

Figura 35 – Algoritmo de carga 1 para MongoDB

```
1
2  $this->iniciar();
3  $numeroControle = $db->getMax('material', 'numerocontrole');
4  for ( $i = 0; $i < $registros; $i++ ) {
5    $documento = $this->obterDocumento( ('' . $numeroControle), $registros,
   $tipoid, $tipofisicoid);
6    $numeroControle++;
7    $db->insert('material', $documento);
8  }
9
10 $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

O código do algoritmo 2 lida com inserções em operações reduzidas. A inserção ocorre em blocos de 10.000 materiais. Para até 10.000, a inserção ocorre em uma única transação. Essa limitação foi imposta pelo MongoDB, pois segundo o manual do MongoDB (2016), a operação de inserção em massa possui a limitação de tratar até 16MB de informação. Na Figura 36 é apresentado o código do algoritmo 2 para PostgreSQL, onde os registros são inseridos em menos operações do que o algoritmo 1.

Figura 36 – Algoritmo de carga 2 para PostgreSQL

```
1
2  $this->iniciar();
3  $iteracoes = [];
4  if ( $registros > 10000 ) {
5    $registrosQuantidade = $registros;
6    $quantidade =  ceil($registrosQuantidade / 10000);
7    for ( $i=0; $i < $quantidade; $i++ ) {
8      if ( $registrosQuantidade > 10000 ){
9        $iteracoes[$i] = 10000;
10       $registrosQuantidade -= 10000;
11     }
12     else {
13       $iteracoes[$i] = $registrosQuantidade; }
14     }
15   }
16   else{
17     $iteracoes[] = $registros;
18   }
19
20 foreach ( $iteracoes as $iteracao ) {
21  $sqls = [];
22  for ( $i = 0; $i < $iteracao; $i++ ) {
23     $numeroControle = ControleMaterial::proximoNumeroDeControlePg();
24
25     // SQL controle material.
26     $sqls[] = $this->obterSQLControle($numeroControle);
27
28     // SQL material.
29     $material = $this->obterSQLMaterial($numeroControle, $registros);
30     $sqls[] = $material[0];
31     …
32     $sqls[] = $material[9];
33
34      // SQL controle exemplar.
35     $exemplar = $this->obterSQLExemplar($numeroControle);
36     $sqls[] = $exemplar[0];
37      ...
38     $sqls[] = $exemplar[9];
39   }
40
41   $sql = implode("\n", $sqls);
42   $db->execute($sql);
43 }
44 $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

O código do algoritmo 2 para o MongoDB é apresentado na Figura 37, onde os documentos são acumulados na variável "$documentos" e inseridos a cada 10.000 documentos.

Figura 37 – Algoritmo de carga 2 para MongoDB

```
1
2  $this->iniciar();
3  $numeroControle = $db->getMax('material', 'numerocontrole');
4
5  if ( $registros > 10000 ){
6    $registrosQuantidade = $registros;
7    $quantidade =  ceil($registrosQuantidade / 10000);
8    for ( $i=0; $i < $quantidade; $i++ ){
9      if ( $registrosQuantidade > 10000 ){
10       $iteracoes[$i] = 10000;
11       $registrosQuantidade -= 10000;
12     }
13     else{
14       $iteracoes[$i] = $registrosQuantidade;
15     }
16   }
17 }
18 else{
19   $iteracoes[] = $registros;
20 }
21
22 foreach ( $iteracoes as $iteracao ){
23   $documentos = [];
24   for ( $i = 0; $i < $iteracao; $i++ ){
25     $documentos[] = $this->obterDocumento(('' . $numeroControle), $registros,
   $tipoid, $tipofisicoid, $tipo);
26     $numeroControle++;
27   }
28
29   $db->batchInsert('material', $documentos);
30 }
31 $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

Conforme método estatístico apresentado no capítulo anterior, o teste de carga foi executado para inserir blocos com 21, 210, 2.100, 105.000, 210.000, 525.000 e 1.050.000 registros. O teste de carga é composto por dois algoritmos, o primeiro executa as inserções em transações distintas e o segundo em menos transações (em blocos de 10.000).

Os algoritmos foram executados 10 vezes cada, sendo que 8 resultados foram considerados. Os resultados da execução do primeiro algoritmo são apresentados na Tabela 9.

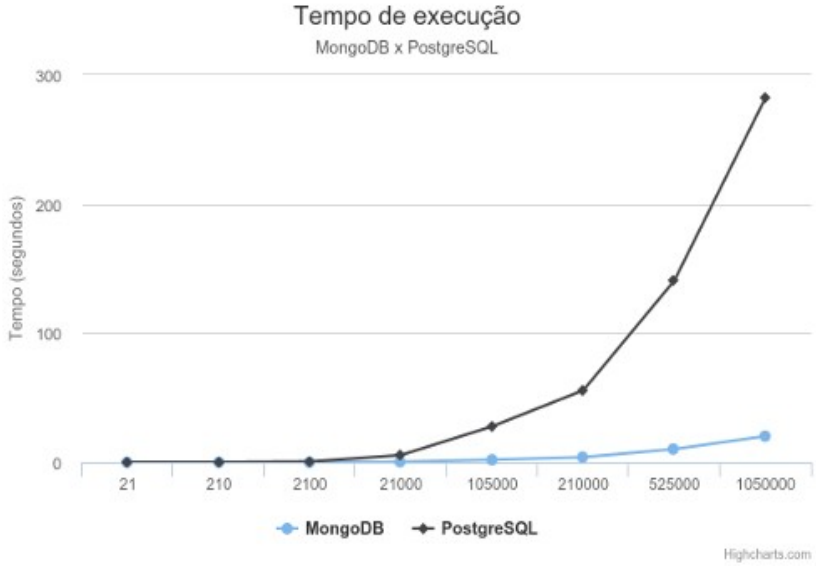
Tabela 9 – Resultado do algoritmo 1 de carga

Registros	Tempo Postgres (segundos)	Tamanho Postgres (MB)	Tempo MongoDB (segundos)	Tamanho Mongo (MB)
21	0,01073650	5,61	0,00124813	64,00
210	0,05987125	5,64	0,00259163	64,00
2100	0,54976725	5,85	0,03870838	64,00
21000	5,61308588	7,65	0,40505600	64,00
105000	27,85040175	15,73	1,97790613	64,00
210000	55,75868613	27,24	3,99560638	64,00
525000	140,84207488	55,06	10,24981775	192,00
1050000	282,34767988	106,35	20,27090775	448,00

Fonte: Elaborado pelo autor (2016).

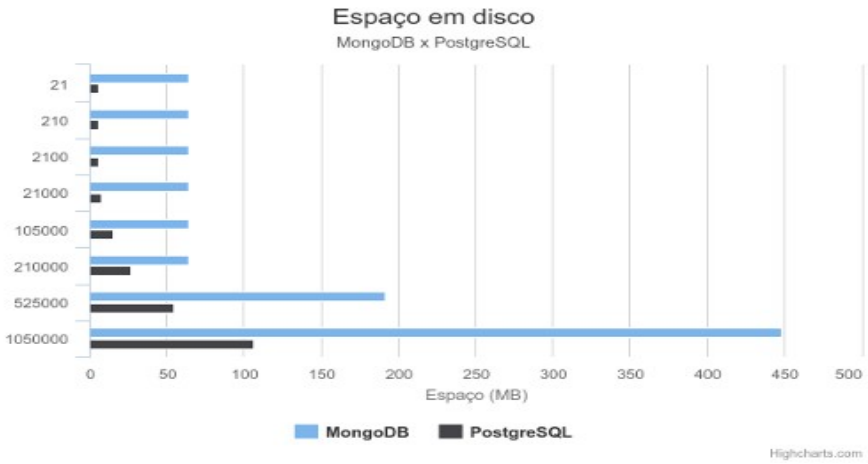
Os resultados apontam que o MongoDB teve performance superior ao PostgreSQL na execução do primeiro algoritmo em relação ao tempo, o resultado é apresentado graficamente na Figura 38. Quanto a utilização de disco, pode ser observado que o armazenamento utilizado permanece inalterado em algumas linhas da tabela. Isso ocorre devido ao fato do MongoDB alocar espaço em disco para garantir que os dados sejam armazenados contiguamente. A utilização de espaço em disco também é apresentada na Figura 39.

Figura 38 – Gráfico de resultados do teste de carga 1



Fonte: Elaborado pelo autor (2016).

Figura 39 – Utilização de espaço em disco do teste de carga 1



Fonte: Elaborado pelo autor (2016).

Os resultados da execução do segundo algoritmo são apresentados na Tabela 10.

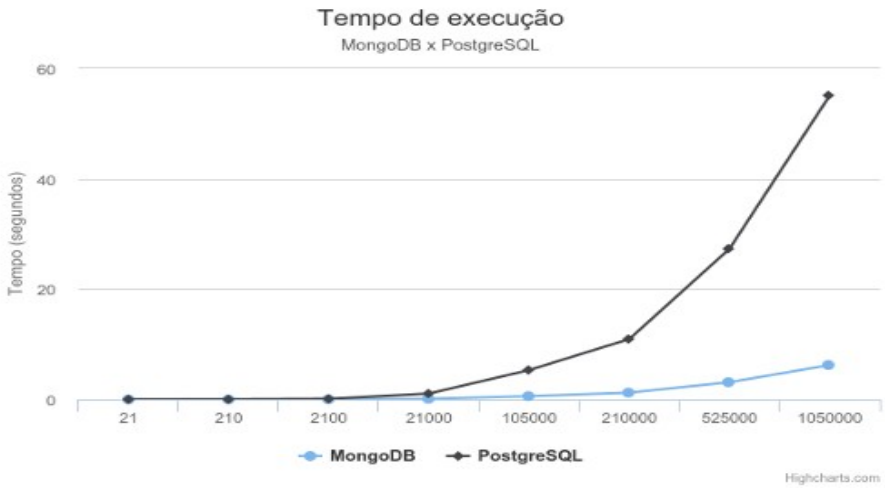
Tabela 10 – Resultado do algoritmo 2 de carga

Registros	Tempo Postgres (segundos)	Tamanho Postgres (MB)	Tempo MongoDB (segundos)	Tamanho Mongo (MB)
21	0,00535163	5,61	0,00099713	64,00
210	0,02144313	5,64	0,00205863	64,00
2100	0,11118550	5,86	0,01834838	64,00
21000	1,07011913	7,66	0,12084838	64,00
105000	5,33054900	16,51	0,59516850	64,00
210000	10,95722800	34,06	1,23429150	64,00
525000	27,32693463	71,19	3,12340500	192,00
1050000	55,12161763	128,16	6,23943988	448,00

Fonte: Elaborado pelo autor (2016).

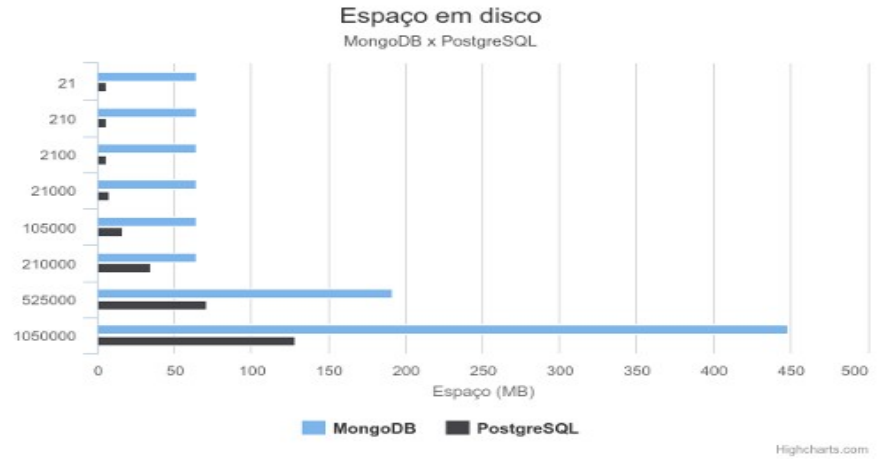
Os resultados demostram que no segundo teste, onde os registros e documentos são inseridos utilizando menos transações que no primeiro algoritmo, o MongoDB novamente apresentou performance superior ao MongoDB em relação ao tempo. Também é possível notar que nesse algoritmo a performance em ambos os SGDB's foi maior em relação ao primeiro algoritmo. O resultado em relação ao tempo é apresentado graficamente na Figura 40 e o armazenamento em disco na Figura 41.

Figura 40 – Gráfico de resultados do teste de carga 2



Fonte: Elaborado pelo autor (2016).

Figura 41 – Utilização de espaço em disco do teste de carga 2



Fonte: Elaborado pelo autor (2016).

Em ambos os testes, o MongoDB apresentou performance superior em relação ao tempo, isso se deve principalmente ao armazenamento em memória virtual e o fato de armazenar dados em apenas uma coleção. Os dados são armazenados em memória virtual, onde a sincronização com o disco rígido ocorre em um intervalo pré-definido. Também é possível perceber que a diferença de tempo entre os SGDB's aumenta conforme aumenta a quantidade de registros, caracterizando que o MongoDB foi projetado para lidar com grande volume de dados.

5.3 Teste de volume

O teste de volume realizou consultas nos dados gerados pelo teste de carga. Na Figura 42 é apresentado o algoritmo aplicado no PostgreSQL, onde são obtidos os registros das tabelas “gtccontrolematerial”, “gtcmaterial” e “gtccontroleexemplar”.

Figura 42 – Algoritmo de volume para PostgreSQL

```
1
2   $this->iniciar();
3
4   // Obtém SQL's.
5   $sqlMaterialControle = $this->obterSQLControle();
6   $sqlMaterial = $this->obterSQLMaterial();
7   $sqlExemplar = $this->obterSQLExemplar();
8
9   // Realiza a execução.
10  $db->execute($sqlMaterialControle);
11  $db->execute($sqlMaterial);
12  $db->execute($sqlExemplar);
13
14  $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

O código do algoritmo aplicado no MongoDB é apresentado na Figura 43, onde são retornados todos os documentos contidos na coleção “material”.

Figura 43 – Algoritmo de volume para PostgreSQL

```
1
2   $this->iniciar();
3   $db->find('material', []);
4   $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

O teste de volume, seguindo a mesma linha do teste de carga, foi aplicado sobre diferentes quantidades de registros e também utilizou o mesmo método estatístico do capítulo anterior. Os resultados da aplicação do teste são apresentados na Tabela 11.

Tabela 11 – Resultado do teste de volume

Registros	Tempo Postgres (segundos)	Tempo MongoDB (segundos)
21	0,00066363	0,00004863
210	0,00086738	0,00005275
2100	0,00277275	0,00004775
21000	0,02068025	0,00004663
105000	0,10434175	0,00005150
210000	0,19547763	0,00005400
525000	0,45658263	0,00004738
1050000	0,93997550	0,00005313

Fonte: Elaborado pelo autor (2016).

Os resultados apontam uma performance superior do MongoDB em relação ao tempo. É possível notar que o tempo do PostgreSQL aumenta proporcionalmente em relação a quantidade de registros. O mesmo não ocorre para o MongoDB. Isso se deve ao fato do MongoDB utilizar memória virtual e também ter sido executado logo após o teste de carga. Portanto, é provável que todos os dados ainda estivessem em memória RAM. O resultado é apresentado graficamente na Figura 44 .

Figura 44 – Gráfico do teste de volume

Fonte: Elaborado pelo autor (2016).

5.4 Teste de stress

O teste de stress, diferente do teste de carga e volume, é avaliado sobre uma quantidade fixa de registros, variando somente a quantidade simultânea de acessos ao banco

de dados. Na Figura 45 é apresentado o algoritmo utilizado no PostgreSQL, ele é executado em fluxos de execução que ocorrem de forma paralela.

Figura 45 – Algoritmo de volume para PostgreSQL

```
1
2  $this->iniciar();
3  $testeStress = new TesteStressPostgres();
4  $testeStress->algoritmo1();
5  $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

O código do algoritmo aplicado no MongoDB é apresentado na Figura 46, ele é muito semelhante ao anterior (para PostgreSQL), pois também invoca o teste de volume. Nesse caso o algoritmo também é executado em fluxos de execução que ocorrem de forma paralela.

Figura 46 – Algoritmo de volume para PostgreSQL

```
1  $this->iniciar();
2  $testeStress = new TesteStressMongo();
3  $testeStress->algoritmo1();
4  $tempo = $this->finalizar();
```

Fonte: Elaborado pelo autor (2016).

Os testes foram executados sobre a quantidade de 1.050.000 registros e são apresentados na Tabela 12.

Tabela 12 – Resultado do teste de stress

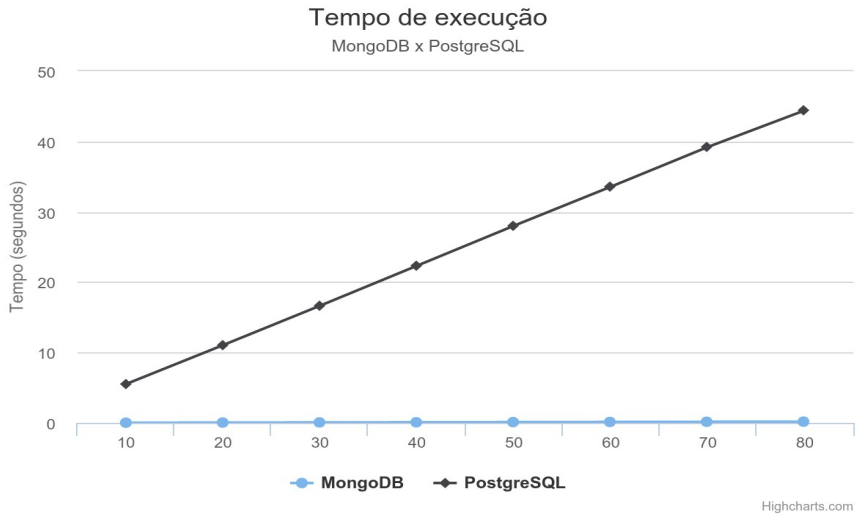
Acessos simultâneos	Tempo Postgres (segundos)	Tempo MongoDB (segundos)
10	5,51252163	0,02354875
20	11,04437588	0,04398850
30	16,64760738	0,06530563
40	22,33245963	0,08693125
50	28,01788775	0,10835313
60	33,57417300	0,13103038
70	39,22137450	0,15211425
80	44,42529413	0,17479413

Fonte: Elaborado pelo autor (2016).

Os resultados apresentados mostram que o MongoDB novamente apresentou performance superior em relação ao PostgreSQL. O teste de stress executa o teste de volume

simultaneamente, começando com 10 acessos até 80. O resultado é apresentado graficamente na Figura 47.

Figura 47 – Gráfico do teste de stress



Fonte: Elaborado pelo autor (2016).

5.5 Teste de tolerância a falhas

O PostgreSQL possui controle transacional, portanto a inserção de registros nesse SGDB foi executada entre os comandos “begin” e “commit”, conforme Figura 48.

Figura 48 – Algoritmo de tolerância a falhas para PostgresSQL

```
1   Pg::begin();
2   for ( $i = 0; $i < $registros; $i++ ){
3     $numeroControle = ControleMaterial::proximoNumeroDeControlePg();
4     $sqlControle = $this->obterSQLControle($numeroControle);
5     $db->execute($sqlControle);
6     $sqlsMaterial = $this->obterSQLMaterial($numeroControle, $registros);
7
8     foreach ( $sqlsMaterial as $sql ) {
9         $db->execute($sql);
10    }
11    $sqlsExemplar = $this->obterSQLExemplar($numeroControle);
12    foreach ( $sqlsExemplar as $sql ) {
13      $db->execute($sql);
14    }
15
16    if ( $pontoFalha != $registros && ($i+1) == $pontoFalha ) {
17      Pg::rollback();
18      exit();
19    }
20  }
21  Pg::commit();
```

Fonte: Elaborado pelo autor (2016).

Pelo fato do MongoDB não apresentar controle transacional, as suas operações não são encapsuladas por um controle do SGDB que possa reverter operações em caso de falhas. O algoritmo é apresentado na Figura 49.

Figura 49 – Algoritmo de tolerância a falhas para PostgresSQL

```
1
2   $numeroControle = $db->getMax('material', 'numerocontrole');
3   for ( $i = 0; $i < $registros; $i++ ) {
4     $documento = $this->obterDocumento( ('' . $numeroControle), $registros,
    $tipoid, $tipofisicoid);
5     $numeroControle++;
6     $db->insert('material', $documento);
7
8     if ( $pontoFalha != $registros && ($i+1) == $pontoFalha ) {
9       exit();
10    }
11  }
```

Fonte: Elaborado pelo autor (2016).

O teste de tolerância a falhas não utilizou o método estatístico apresentado no capítulo anterior, portanto seus resultados foram validados em uma única execução. Os resultados são apresentados na Tabela 13, a coluna “Quantidade de materiais” apresenta a quantidade de registros que o teste tentou inserir, a coluna “Ponto de falha” apresenta a quantidade de materiais que foram inseridos antes da falha, a coluna “Total PostgreSQL” apresenta o total de materiais após a falha no PostgreSQL e “Total MongoDB” para o MongoDB .

Tabela 13 – Resultado do teste de tolerância a falhas

Quantidade de materiais	Ponto de falha	Total PostgreSQL	Total MongoDB
1000	998	0	998
100	50	0	50
10000	8000	0	8000
1000	1000	1000	1000

Fonte: Elaborado pelo autor (2016).

Os resultados apresentam a diferença de quantidade de registros após uma falha em cada banco de dados. O PostgreSQL possui controle transacional, portanto, após uma falha durante a inserção de registros, o banco é capaz de retornar ao estado anterior. O MongoDB não possui controle transacional, portanto, os registros que foram inseridos antes da falha não são revertidos. Os resultados evidenciam as propriedades ACID no PosgreSQL, e, para obter o mesmo resultado no MongoDB, o controle transacional deve ser resolvido a nível de

aplicação. Na última linha da tabela não foi executado a falha, portanto, em ambos SGDB's foram inseridos todos os registros.

5.6 Impedância

Segundo Almeida (2015), a impedância é a diferença estrutural entre o banco de dados utilizado e os dados e estruturas em memória. Para o autor, a carga de trabalho de desenvolvimento de um sistema é proporcional a impedância. Quanto menor ela for, menos conversões devem ser feitas para armazenar os dados.

Seguindo essa linha de pensamento, a impedância do PostgreSQL e do MongoDB foi avaliada durante o desenvolvimento do protótipo e foi classificada como impedância de estrutura e impedância de código fonte.

5.6.1 Impedância na estrutura

Para armazenar materiais no PostgreSQL, foram necessárias três tabelas para contemplar o armazenamento dinâmico de informações, a tabela “gtccontrolematerial” armazena dados de controle como a data de entrada, o número de controle, tipo, tipo físico e data de alteração do material, a tabela “gtccontroleexemplar” é responsável pelo armazenamento dos exemplares, e a tabela “gtcmaterial” armazena os metadados MARC (título, autor e outras informações).

No MongoDB, para armazenar o mesmo material é necessário apenas um documento, composto por dados de controle, exemplares e metadados. Isso evidencia que o MongoDB é mais apropriado para armazenar registros bibliográficos em MARC, pois eles foram armazenados em uma única estrutura, sem a necessidade de particionar os dados, facilitando a inserção e a busca.

5.6.2 Impedância no código fonte

A implementação do protótipo não utilizou *framework* de persistência de dados, isso foi necessário para avaliar a performance dos SGDB's dentro da aplicação, pois, ao utilizar um *framework* de persistência, detalhes além do SGDB devem ser observados. Portanto, foram utilizados os recursos nativos do PHP como o *pg_connection, pg_exec, pg_query e pg_fetch_assoc* para o PostgreSQL e *find, insert, update, findOne* e *execute* da classe MongoClient para o MongoDB. A fim de facilitar o desenvolvimento, foram criadas as classes Pg e MongoDB, cujo código fonte se encontra no Apêndice A deste trabalho. Para inserir dados no PostgreSQL, os dados armazenados em memória tiveram de ser traduzidos em comandos SQL. A Figura 50 apresenta a inserção de um tipo de material a partir de um *array* com os dados.

Figura 50 – Método para inserir tipo de material no PostgreSQL

```
1  public static function insertPg($data) {
2  $db = Pg::getInstance();
3  $sql = "INSERT INTO gtcmaterialtipo VALUES (" . $data["id"] . ",'" .
   $data["descricao"] . "','" . $data["observacao"] . "');";
4
5     return $db->execute($sql);
6  }
```

Fonte: Elaborado pelo autor (2016).

Para o MongoDB não foi necessário realizar nenhum tipo de conversão dos dados. A Figura 51 apresenta a inserção do mesmo *array* de dados.

Figura 51 – Método para inserir tipo de material no MongoDB

```
1  public static function insertMongo($data){
2    $db = MongoDB::getInstance();
3    return $db->insert('tipo', $data);
4  }
```

Fonte: Elaborado pelo autor (2016).

Todas as operações de dados como inserir, editar, excluir e buscar no PostgreSQL devem ser feitas a partir de comandos SQL, portanto, todas as operações tiveram de ser traduzidas nesses comandos. Para realizar essas mesmas operações no MongoDB, nenhuma conversão foi necessária. Isso evidencia que o MongoDB possui impedância menor, e é mais indicado para o desenvolvimento de sistemas em PHP.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma análise comparativa dos Sistemas Gerenciadores de Banco de Dados PostgreSQL (modelo relacional) e MongoDB (NoSQL) de forma qualitativa e quantitativa, quanto ao desempenho e flexibilidade na busca e armazenamento de registros bibliográficos MARC. A análise foi realizada através do desenvolvimento de um protótipo para realizar ações básicas de um sistema gerenciador de acervo de bibliotecas, como importar e gerenciar materiais e para realização de testes de busca, carga, volume, stress e tolerância a falhas.

O teste de tolerância a falhas demonstra que o PostgreSQL implementa as propriedades ACID do modelo relacional, portanto o sistema é capaz de retornar ao estado anterior após a ocorrência de uma falha. Esse recurso não está presente no MongoDB, fazendo com que esse recurso tenha que ser desenvolvido na aplicação. Ele não se fez necessário no protótipo, mas para um sistema em produção seria necessário o seu desenvolvimento.

O teste de busca utilizou duas estratégias com objetivo de analisar separadamente o custo do banco de dados para realizar uma busca e o custo que a aplicação possui ao obter os dados. A primeira estratégia, onde todos os resultados da busca são obtidos o PostgreSQL demonstrou uma performance superior ao MongoDB, isso aconteceu porque o primeiro carrega o resultado em memória e retorna um ponteiro, o segundo retorna um cursor. Obter e iterar sobre dados na memória é mais rápido do que operar um cursor do MongoDB. A segunda estratégia, onde os dados são apenas contados, o MongoDB apresentou uma performance superior. Os resultados apontam que a nível de banco, o MongoDB possui

performance superior de busca de registros MARC. A obtenção de dados, a nível de aplicação, pode ser resolvida através da implementação de paginação de resultados, recurso que não foi necessário no protótipo.

O teste de carga, volume e stress demonstram que o MongoDB possui performance superior ao PostgreSQL, isso se deve principalmente ao fato do registro bibliográfico ser armazenado em apenas um documento, enquanto para armazenar um registro completo com todas as etiquetas MARC no PostgreSQL são necessários múltiplos registros, em tabelas diferentes. Outro fato relevante para essa diferença é o uso de memória virtual, onde os dados são armazenados em memória e depois são sincronizados para o disco. Os testes também demonstram que apesar do registro ser armazenado em apenas um documento no MongoDB, o uso de espaço em disco não foi inferior ao PostgreSQL.

Além de uma avaliação quantitativa, através da execução de testes de performance, foi realizada uma análise qualitativa da impedância de ambos SGDB's, que avaliou a compatibilidade dos mesmos para o armazenamento de registros bibliográficos em MARC. O MongoDB possui uma compatibilidade superior ao PostgreSQL, tanto a nível de estrutura de armazenamento quanto a nível de código fonte, pois para armazenar um registro no PostgreSQL são necessárias três tabelas, enquanto no MongoDB é necessário apenas uma coleção. Essa diferença é refletida no código fonte, onde no primeiro são necessárias um número superior de conversões do registro, para que o mesmo seja armazenado múltiplas linhas e em múltiplas tabelas.

A construção do protótipo e obtenção dos resultados a partir dele evidenciou o cumprimento tanto dos objetivos primários, quanto dos secundários propostos, pois a compreensão das características do modelo relacional e NoSQL foi fundamental para o desenvolvimento da análise.

Os resultados obtidos através da execução dos testes apontam que o MongoDB, apesar de não possuir tolerância a falhas nativa, é fortemente indicado para sistemas de gestão de acervos de bibliotecas que utilizam o modelo de metadados MARC, apresentando uma performance superior ao PostgreSQL na busca, armazenamento de registros e compatibilidade estrutural, resolvendo de maneira nativa o armazenamento flexível de registros bibliográficos.

Como trabalhos futuros pretende-se a ampliação do protótipo com as seguintes funcionalidades:

a) ser utilizado para testar outros SGDB's: o protótipo desenvolvido possui métodos e classes distintas para cada sistema, sendo necessário pouco esforço para contemplar um novo SGDB;

b) realizar teste de concorrência entre usuários: ampliar o protótipo para testar a concorrência entre usuários através das funcionalidades de empréstimos e reservas de exemplares.

REFERÊNCIAS

ALMEIDA, Flávio. **MEAN: Full stack Javascript para aplicações web com MongoDB, Express, Angular e Node**. 1 ed. São Paulo: Casa do Código: 2014.

ALMEIDA, Luís Fernando Barbosa. **A metodologia de Disseminação da Informação Geográfica e os Metadados**. 1999. 201 f. Tese (Doutorado) – Doutorado em Geográfia, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 16 mar. 1999. Disponível em: <http://www.cprm.gov.br/publique/media/dou_fernandobarbosa.pdf>. Acesso em: 12 ago. 2015.

ALVES, Maria das Dores Rosa; SOUZA, Marcia Izabel Fugisawa. Estudo de correspondência de elementos metadados: DUBLIN CORE e MARC 21. **Revista Digital de Biblioteconomia e Ciência da Informação**, Campinas, v. 4, n. 2, 2007. Disponível em: <http://www.sbu.unicamp.br/seer/ojs/index.php/rbci/article/view/358/237>. Acesso em: 12 ago. 2015.

BARBOSA, Elvina Maria de Sousa; EDUVIRGES, Joelson Ramos. O Formato Marc 21: principais vantagens para bibliotecários, bibliotecas e usuários para a recuperação da informação. **Encontro Nacional de Estudantes de Biblioteconomia, Documentação, Gestão e Ciência da Informação**. 21., 2010. Paraíba. 2010; Disponível em: <http://rabci.org/rabci/sites/default/files/O%20FORMATO%20MARC%2021%20principais%20vantagens%20para%20bibliotec%C3%A1rios,%20bibliotecas%20e%20usu%C3%A1rios%20para%20a%20recupera%C3%A7%C3%A3o%20da%20informa%C3%A7%C3%A3o_0.pdf>. Acesso em: 29 set. 2015.

BOAGLIO, Fernando. **MongoDB: Construa novas aplicações com novas tecnologias**. 1. ed. São Paulo: Casa do Código, 2015.

CHANG, Fay et al. Bigtable: A Distribuited Storage System for Structured Data. **Symposium on Operating System Design and Implementation**. 7., 2006, Seattle, 2006. Disponível em: <http://static.googleusercontent.com/media/research.google.com/pt-BR//archive/bigtable-osdi06.pdf>. Acesso em: 02 set. 2015.

CHEMIN, Beatris Francisca. **Manual da Univates para trabalhos acadêmicos:** planejamento, elaboração e apresentação. 3. ed. Lajeado: Editora Univates, 2015. E-book. Disponível em: <https://www.univates.br/editora-univates/media/publicacoes/110/pdf_110.pdf>. Acesso em: 06 out. 2015.

CHODOROW, Kristina; DIROLF, Michael. MongoDB: The Definitive Guide. 1. ed. Sebastopol: O'Relly Media, 2010.

CORTÊ, Adelaide Ramos et al. Automação de bibliotecas e centros de documentação: o processo de avaliação e seleção de softwares. **Revista Ci. Inf**, Brasília, v. 28, n. 3, p.241-256, set/dez. 1999. Disponível em <http://www.scielo.br/pdf/%0D/ci/v28n3/v28n3a2.pdf>. Acesso em 30 mai. 2016.

DATE, C. J. **Introdução a sistemas de bancos de dados**. 8 ed. Rio de Janeiro: Campus, 1990.

DIANA, Maurício de, GEROSA, Marco Aurélio. NOSQL na Web 2.0: Um Estudo Comparativo de Bancos Não-Relacionais para Armazenamento de Dados na Web 2.0. **Workshop de Teses e Dissertações em Banco de Dados**, 9., 2010, Belo Horizonte: UFMG. 2010. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/wtdbd/2010/sbbd_wtd_12.pdf>. Acesso em: 13 ago. 2015.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados**. 4. ed. São Paulo: Pearson Addison Wesley, 2005.

FILHO, Marcos André Pereira Martins. **SQL x NOSQL:** Análise de Desempenho Do Uso Do MongoDB Em Relação Ao Uso do PostgreSQL. 2015. 54 p. Monografia (Graduação) – Curso de Ciência da Computação, Universidade Federal de Pernambuco, fev. 2015. Disponível em: <http://www.cin.ufpe.br/~tg/2014-2/mapmf.pdf>. Acesso em: 14 ago. 2015.

FUHR, Bruno Edgar. **Desenvolvimento de uma Ferramenta de Coleta e Armazenamento De Dados Para Big Data**. 2014. 82 p. Monografia (Graduação) – Curso de Sistemas de Informação, UNIVATES, Lajeado, nov. 2014. Disponível em: <https://www.univates.br/bdu/bitstream/10737/651/1/2014BrunoEdgarFuhr.pdf>. Acesso em: 14 ago. 2015.

FURRIE, B. **Understanding MARC bibliographic:** machine-readable cataloging. 7 ed. rev. Washington, D. C.: Library of Congress; Follet Software, 2003. Disponível em: <http://www.loc.gov/marc/umb>. Acesso em: 14 set. 2015.

GIL, Antônio C. **Como elaborar projetos de pesquisa**. 4. ed. São Paulo: Atlas, 2006.

GIL-LEIVA, Isidoro. A indexação na internet. **Brazilian Journal of Information Science**, v. 1, n. 2, p. 47-68, jul./dez. 2007. Disponível em: <http://www2.marilia.unesp.br/revistas/index.php/bjis/article/viewFile/38/36>. Acesso em: 11 ago. 2015.

HEWITT, E. Cassandra: **The Definitive Guide**. Sebastopol: O'Reilly Media, 2011.

JSON. **Introdução ao JSON**. Disponível em: <http://json.org/json-pt.html>. Acesso em: 15 ago. 2015.

LEOPARDI, Maria T. **Metodologia da pesquisa na saúde**. 2. ed. Florianópolis: UFSC, 2002.

LÓSCIO, Bernadette Farias; OLIVEIRA, Hélio Rodrigues de; PONTES, César de Sousa. NoSQL no desenvolvimento de aplicações Web colaborativas. **Simpósio Brasileiro de Sistemas Colaborativos**, 7., 2011, Paraty, 2011. Disponível em: <http://www.addlabs.uff.br/sbsc_site/SBSC2011_NoSQL.pdf>. Acesso em: 13 ago. 2015.

MALHOTRA, Naresh K. **Pesquisa em marketing:** uma orientação aplicada. 4. ed. Porto Alegre: Bookman, 2006.

MEZZAROBA, Orides; MONTEIRO, Cláudia S. **Manual de metodologia da pesquisa no Direito**. 3. ed. São Paulo: Saraiva, 2006 .

MILANI, A. **PostgreSQL:** Guia do Programador. 1. ed. São Paulo: Novatec Editora, 2008.

MONIRUZZAMAN, A. B. M.; HOSSAIN, Syed Akhter. NoSQL Database: New Era of Databases for Big Data Analytics – Classification, Characteristics and Comparison. **International Journal of Database Theory and Application**. v. 6, n. 4, 2013. Disponível em: <http://arxiv.org/pdf/1307.0191v1.pdf>. Acesso em: 16 ago. 2015.

MONGODB. **Aggregation**. Disponível em: <https://docs.mongodb.org/manual/core/aggregation-introduction>. Acesso em: 05 set 2015.

MONGODB. **Index Types**. Disponível em: <https://docs.mongodb.org/manual/core/index-types/>. Acesso em: 05 set 2015.

MONGODB. **Journaling and MMAPv1**. Disponível em: <https://docs.mongodb.org/v3.0/core/journaling/#journaling-and-mmapv1>. Acesso em: 05 abr. 2016.

MONGODB. **MMAPv1 Storage Engine**. Disponível em: <https://docs.mongodb.org/v3.0/core/mmapv1/>. Acesso em: 05 abr. 2016.

MONGODB. **MongoDB Limits and Thresholds**. Disponível em: <https://docs.mongodb.org/manual/reference/limits/>. Acesso em: 04 abr. 2016.

NISO. **Understanding Metadata**. Bethesda: NISO Press, 2004. Disponível em: <http://www.niso.org/standards/resources/UnderstandingMetadata.pdf>. Acesso em: 11 ago. 2015.

PHP. **Pg_query**. Disponível em: <http://php.net/pg_query>. Acesso em: 07 abr. 2016.

PHP. **The MongoCursor class** . Disponível em: <http://php.net/manual/pt_BR/class.mongocursor.php>. Acesso em: 07 abr. 2016.

POLITOWSKI, Cristino; MARAN, Vinícius. Comparação de Performance entre PostgreSQL e MongoDB. **ERBD**. 10., 2014, São Francisco do Sul, 2014. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erbd/2014/003.pdf>. Acesso em 15 set. 2015.

POSTGRESQL. Documentation. Disponível em: <http://www.postgresql.org/docs/manuals/archive/> Acesso em: 18 set. 2015.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de Gerenciamento de Banco de Dados**. 3. ed. São Paulo: McGraw-Hill, 2008.

REDMOND, Eric; WILSON, Jim R. **Seven Databases in Seven Weeks:** A Guide to Modern Databases and the NoSQL Movement. 1. ed. Dallas: Pragmatic Programmers, 2012. Disponível em: <http://www.pdfiles.com/pdf/files/English/Databases/Seven_Databases_In_Seven_Weeks.pdf>. Acesso em: 03 set 2015.

RIBEIRO, Antonia Motta de Castro Memória. **Catalogação de recursos bibliográficos:** AACR2R em MARC 21. 3 ed. Brasília: Memória Ribeiro, 2006.

ROCHA, Rafael Port da. Metadados, Web Semântica, Categorização Automática: combinando esforços humanos e computacionais para descoberta e uso dos recursos da web. **Em Questão**, Porto Alegre, v. 10, n. 1, p. 109-121, jan./jul. 2004. Disponível em: <http://www.brapci.ufpr.br/download.php?dd0=9814>. Acesso em: 12 ago. 2015.

ROSA, Adriano Guzzo. **Análise comparativa do banco de dados MongoDB:** Testes de desempenho MongoDB x MSQL. 2013. 78 p. Monografia (Graduação) – Curso de Ciência da Computação, Universidade Vila Velha, Vila Velha, 2013. Disponível em: <https://play.google.com/books/reader?printsec=frontcover&output=reader&id=ShpFBQAAQBAJ>. Acesso em: 15 set 2015.

SETZER, Valdemar W. Dado, Informação, Conhecimento e Competência. **DataGramaZero – Revista de Ciência da Informação**, São Paulo: n. 0, 1999. Disponível em: <http://www.ime.usp.br/~vwsetzer/datagrama.html>. Acesso em: 28 set. 2015.

SILBERSCHATZ, Abraham; KORTH, Henry F., SUDARSHAN, S. **Sistema de Banco de Dados**. 3. ed. São Paulo: Pearson Makron Books, 1999.

THE APACHE SOFTWARE FOUNDATION. **What is Apache Cassandra?**. Disponível em: <http://www.planetcassandra.org/what-is-apache-cassandra>. Acesso em: 01 set. 2015.

TIWARI, Shashank. **Professional NoSQL**. 1 ed. Indianapolis: John Wiley & Sons, 2011.

VALMORBIDA, Willian. **Análise e implementação de um sistema integrado de busca baseado nos padrões de metadados e protocolos de interoperabilidade utilizados por catálogos on-line de bibliotecas e repositórios digitais**. 2011. 115 p. Monografia (Graduação) – Curso de Sistemas de Informação, UNIVATES, Lajeado, nov. 2011. Disponível em: <https://www.univates.br/bdu/bitstream/10737/249/1/WillianValmorbida.pdf>. Acesso em: 17 set. 2015.

VAZ, Maria Salete Marcon Gomes. **MetaMídia – Um Modelo de Metadados na Indexação e Recuperação de Objeto Multimídia**. 2000. 142 f. Tese (Doutorado) – Doutorado em Ciência da Computação, Universidade Federal de Pernambuco, Recife, dez. 2000. Disponível em: <http://ri.uepg.br:8080/riuepg/bitstream/handle/123456789/638/TESE_MariaSaleteMarconGomesVaz.pdf?sequence=1>. Acesso em: 12 ago. 2015.

VETTER, Silvana Maria de Jesus; ARAUJO, Leonardo Pinto. PADRÃO MARC 21 E CATALOGAÇÃO EM BIBLIOTECAS UNIVERSITÁRIAS DE SÃO LUÍS/MA. **Seminário Nacional de Bibliotecas Universitárias**, Gramado, 2012. Disponível em: <http://www.snbu2012.com.br/anais/pdf/4RJ6.pdf>. Acesso em 13 ago. 2015.

VIEIRA, Marcos Rodrigues; FIGUEIREDO, Josiel Maimone de; LIBERATTI, Gustavo; VIEBRANTZ, Alvaro Fellipe Mendes. Bancos de Dados NoSQL: Conceitos, Ferramentas, Linguagens e Estudos de Casos no Contexto de Big Data. **Simpósio Brasileiro de Bancos de Dados**. 27., 2012, São Paulo. Disponível em: <http://data.ime.usp.br/sbbd2012/artigos/pdfs/sbbd_min_01.pdf>. Acesso em 01 set. 2015.

APÊNDICES

APÊNDICE A – Código fonte completo das classes Pg e MongoDB

```
1
2   <?php
3
4   namespace App;
5
6   class Pg {
7       public $connection = NULL;
8       static private $instance = NULL;
9
10      public static function getInstance(){
11          if (self::$instance == NULL){
12              self::$instance = new Pg();
13          }
14
15          return self::$instance;
16      }
17
18      public function __construct(){
19          $this->connect();
20      }
21
22      public function connect() {
23          $config = postgres();
24
25          $this->connection = pg_connect('host=' . $config->host . ' port=' .
    $config->port . ' dbname=' . $config->name . ' user=' . $config->user . '
    password=' . $config->password);
26      }
27
28      public function execute($sql){
29          try {
30              return pg_exec($this->connection, $sql);
31          }
32          catch( Exception $e ) {
33              die($e->getMessage());
34          }
35      }
36
37
38      public static function obterProximoId($sequencia){
39          $db = Pg::getInstance();
40
41          $sql = "select nextval('{$sequencia}') as proximo";
42          $result = $db->query($sql);
43
44          if ( count($result) > 0 ) {
45              return $result[0]->proximo;
46          }
47          eles {
48              return null;
49          }
50      }
51
52      public function query($sql, $retornaObjeto=true){
53          try {
54              $query = pg_query($this->connection, $sql);
55              $result = [];
56
57              if ( $query ){
58                  while( $linha = pg_fetch_assoc($query) ){
59                      if ( $retornaObjeto ){
60                          $result[] = (object) $linha;
61                      }
62                      eles {
63                          $result[] = $linha;
```

```
64                      }
65                  }
66              }
67              return $result;
68          }
69          catch( Exception $e ){
70              die($e->getMessage());
71          }
72      }
73
74      public static function begin() {
75          $db = self::getInstance();
76          return $db->execute("BEGIN;");
77      }
78
79      public static function commit(){
80          $db = self::getInstance();
81          return $db->execute("COMMIT;");
82      }
83
84      public static function rollback() {
85          $db = self::getInstance();
86          return $db->execute("ROLLBACK;");
87      }
88
89      public function getDatabaseSize() {
90          $config = postgres();
91          $baseDeDados = $config->name;
92          $size = 0;
93
94          $sql = "select (pg_database_size('{$baseDeDados}') / 1024.0) / 1024.0 as
    tamanho";
95          $result = $this->query($sql);
96
97          if ( count($result) ) {
98              $size = $result[0]->tamanho;
99          }
100         return $size;
101     }
102 }
103
104 ?>
105
106 <?php
107 namespace App;
108
109 class MongoDB {
110     public $mongoClient = null;
111     public $connection = NULL;
112     static private $instance = NULL;
113
114     public static function getInstance(){
115         if (self::$instance == NULL){
116             self::$instance = new MongoDB();
117         }
118         return self::$instance;
119     }
120
121     public function __construct(){
122         $this->connect();
123     }
124
125     public function connect(){
126         $config = mongodb();
127
128         $connecting_string =  sprintf('mongodb://%s:%d/%s', $config->host,
    $config->port, $config->name);
129         $this->mongoClient = new \MongoClient($connecting_string);
```

```
130         //$mongo = new \MongoClient();
131         $this->connection = $this->mongoClient->tcc;
132     }
133 
134     public function query($collection, $filtro=[]){
135         try {
136             $cursor =  $this->connection->{$collection}->find($filtro);
137             $documents = [];
138 
139             foreach ($cursor as $document){
140                 $documents[] = (object) $document;
141             }
142             return $documents;
143         }
144         catch( Exception $e ){
145             die($e->getMessage());
146         }
147     }
148 
149     public function find($collection, $filtro=[]){
150         try {
151             return $this->connection->{$collection}->find($filtro);
152         }
153         catch( Exception $e ){
154             die($e->getMessage());
155         }
156     }
157 
158     public function count($collection, $filtro=[]) {
159         try {
160             return $this->connection->{$collection}->find($filtro)->count();
161         }
162         catch( Exception $e ) {
163             die($e->getMessage());
164         }
165     }
166 
167     public function getMax($collection, $field) {
168         $cursor = $this->connection->material->aggregate(array( '$group' =>
    array( '_id' => '', 'numerocontrole' => array('$max'=>'$numerocontrole'))));
169 
170         $retorno = NULL;
171         if ( count($cursor['result']) > 0 ){
172             $retorno = $cursor['result'][0][$field];
173         }
174         if ( !$retorno ) {
175             $retorno = 2;
176         }
177         return $retorno;
178     }
179 
180     public function insert($collection, $registro) {
181         try {
182             return $this->connection->{$collection}->insert($registro);
183         }
184         catch( Exception $e ){
185             die($e->getMessage());
186         }
187     }
188 
189     public function batchInsert($collection, $registros) {
190         try {
191             return $this->connection->{$collection}->batchInsert($registros);
192         }
193         catch( Exception $e ) {
194             die($e->getMessage());
195         }
196     }
```

```
197
198     public function update($collection, $filtro, $dados){
199         try {
200             return $this->connection->{$collection}->update($filtro, ['$set' =>
    $dados]);
201         }
202         catch( Exception $e ){
203             die($e->getMessage());
204         }
205     }
206
207     public function get($collection, $filtro) {
208         try {
209             return (object) $this->connection->{$collection}->findOne($filtro);
210         }
211         catch( Exception $e ) {
212             die($e->getMessage());
213         }
214     }
215
216     public function delete($collection, $filtro){
217         try {
218             return $this->connection->{$collection}->remove($filtro);
219         }
220         catch( Exception $e ){
221             die($e->getMessage());
222         }
223     }
224
225     public function createReferenceByObjectId($collection, $objectId) {
226         if ( !is_object($objectId) ){
227             $objectId = new \MongoId($objectId);
228         }
229         return $this->connection->createDBRef($collection, $objectId);
230     }
231
232     public function createReferenceById($collection, $id) {
233         $document = $this->connection->{$collection}->findOne(['id' => $id]);
234         return $this->connection->createDBRef($collection, $document['_id']);
235     }
236
237     public function getReference($reference) {
238         return $this->connection->getDBRef($reference);
239     }
240
241     public function getDatabaseSize() {
242         $stats = $this->connection->execute("db.stats()");
243         $size = (int) $stats["retval"]["fileSize"];
244         $size = ($size / 1024) / 1024;
245         return $size;
246     }
247 }
248
249 ?>
```

APÊNDICE B – Código fonte parcial dos testes de performance

```
1
2   <?php
3
4   namespace App\Testes;
5
6   include_once('Testes.php');
7   include_once('../Material.php');
```

```
8  include_once('../MaterialTipo.php');
9  include_once('../MaterialTipoFisico.php');
10 include_once('../MongoDB.php');
11 include_once('../../config/db.php');
12
13 use App\Material;
14 use App\MaterialTipo;
15 use App\MaterialTipoFisico;
16 use App\MongoDB;
17
18 class TesteCargaMongo extends Testes {
19     public function __construct($execucaoId, $url, $iteracoes, $registros,
   $algoritmo='algoritmo1', $apagarRegistros = 0) {
20         parent::__construct($execucaoId, $url);
21
22         $testeArmazenar = new \stdClass();
23         $testeArmazenar->mongo = new \stdClass();
24         $testeArmazenar->mongo->dados = [];
25
26         if ( $apagarRegistros == 1 ) {
27             Material::apagarTudoMongo();
28         }
29
30         $this->iniciar();
31
32         if ( $algoritmo == 'algoritmo1' ) {
33             $testeArmazenar->mongo->tipo = 'carga1';
34             $this->algoritmo1($iteracoes);
35         }
36         else {
37             $testeArmazenar->mongo->tipo = 'carga2';
38             $this->algoritmo2($iteracoes);
39         }
40
41         $tempo = $this->finalizar();
42         $teste = new \stdClass();
43         $teste->registros = $registros;
44         $teste->tempo = $tempo;
45         $teste->tamanhoBase = Material::obterTamanhoBaseMongo();
46         $testeArmazenar->mongo->dados[] = $teste;
47         $this->armazenarTeste($testeArmazenar);
48        }
49
50     public function obterDocumento($numeroControle, $registros, $tipoid,
   $tipofisicoid, $tipo) {
51         $data = date('d/m/Y');
52         $execucaoId = $this->execucaoId;
53
54         $documento = [];
55         $documento['numerocontrole'] = $numeroControle;
56         $documento['numerocontrolepai'] = NULL;
57         $documento['dataentrada'] = $data;
58         $documento['dataultimamudanca'] = $data;
59         $documento['tipoid'] = $tipoid;
60         $documento['tipofisicoid'] = $tipofisicoid;
61
62         if ( $tipo == 1) {
63             $etiqueta1 = [];
64             $etiqueta1['numerocontrole'] = $numeroControle;
65             $etiqueta1['campo'] = '041';
66             $etiqueta1['subcampo'] = 'a';
67             $etiqueta1['linha'] = '0';
68             $etiqueta1['indicador1'] = NULL;
69             $etiqueta1['indicador2'] = NULL;
70             $etiqueta1['conteudo'] = NULL;
71             $etiqueta1['conteudo'] = 'Português';
72             $etiqueta1['conteudopesquisa'] = 'PORTUGUES';
73             // Omitido
```

```
74          }
75          elseif ( $tipo == 2 )
76              // Omitido.
77          elseif ( $tipo == 3 )
78              // Omitido.
79          elseif ( $tipo == 4 )
80              // Omitido.
81          elseif ( $tipo == 5 )
82              // Omitido.
83
84          $documento['etiquetas'] = [$etiqueta1, $etiqueta2, $etiqueta3,
    $etiqueta4, $etiqueta5, $etiqueta6, $etiqueta7, $etiqueta8, $etiqueta9,
    $etiqueta10];
85
86          $exemplar1 = [];
87          $exemplar1['numerocontrole'] = $numeroControle;
88          $exemplar1['numeroitem'] = '1';
89          $exemplar1['tipoid'] = $tipoid;
90          $exemplar1['tipofisicoid'] = $tipofisicoid;
91          $exemplar1['dataentrada'] = $data;
92          $exemplar1['linha'] = '0';
93          $exemplar2 = [];
94          $exemplar3 = [];
95          $exemplar4 = [];
96          $exemplar5 = [];
97          $exemplar6 = [];
98          $exemplar7 = [];
99          $exemplar8 = [];
100         $exemplar9 = [];
101         $exemplar10 = [];
102         // Omitido.
103         $documento['exemplar'] = [$exemplar1, $exemplar2, $exemplar3, $exemplar4,
    $exemplar5, $exemplar6, $exemplar7, $exemplar8, $exemplar9, $exemplar10];
104
105         return $documento;
106     }
107
108       public function algoritmo1($registros) {
109         $db = MongoDB::getInstance();
110         $tipoid = MaterialTipo::obterReferencia('19');
111         $tipofisicoid = MaterialTipoFisico::obterReferencia('13');
112         $numeroControle = $db->getMax('material', 'numerocontrole');
113         $tipo = 1;
114
115               for ( $i = 0; $i < $registros; $i++ ) {
116             $documento = $this->obterDocumento( ('' . $numeroControle),
    $registros,  $tipoid, $tipofisicoid, $tipo);
117
118             if ( $tipo == 5 ) {
119                 $tipo = 1;
120             }
121             else {
122                 $tipo++;
123             }
124             $numeroControle++;
125             $db->insert('material', $documento);
126               }
127       }
128
129     public function algoritmo2($registros) {
130         $db = MongoDB::getInstance();
131
132         $tipoid = MaterialTipo::obterReferencia('21');
133         $tipofisicoid = MaterialTipoFisico::obterReferencia('14');
134         $numeroControle = $db->getMax('material', 'numerocontrole');
135         $tipo = 1;
136         if ( $registros > 10000 ) {
137             $registrosQuantidade = $registros;
```

```
138             $quantidade =  ceil($registrosQuantidade / 10000);
139             for ( $i=0; $i < $quantidade; $i++ ) {
140                 if ( $registrosQuantidade > 10000 ) {
141                     $iteracoes[$i] = 10000;
142                     $registrosQuantidade -= 10000;
143                 }
144                 else {
145                     $iteracoes[$i] = $registrosQuantidade;
146                 }
147             }
148         }
149         else {
150             $iteracoes[] = $registros;
151         }
152 
153         foreach ( $iteracoes as $iteracao ){
154             $documentos = [];
155 
156             for ( $i = 0; $i < $iteracao; $i++ ) {
157                 $documentos[] = $this->obterDocumento(('' . $numeroControle),
    $registros,  $tipoid, $tipofisicoid, $tipo);
158 
159                 if ( $tipo == 5 ) {
160                     $tipo = 1;
161                 }
162                 else {
163                     $tipo++;
164                 }
165 
166                 $numeroControle++;
167             }
168 
169             $db->batchInsert('material', $documentos);
170         }
171     }
172 }
173 
174 ?>
175 
176 <?php
177 
178 namespace App\Testes;
179 
180 include_once('Testes.php');
181 include_once('../ControleMaterial.php');
182 include_once('../ControleExemplar.php');
183 include_once('../Material.php');
184 include_once('../Pg.php');
185 include_once('../../config/db.php');
186 
187 use App\ControleMaterial;
188 use App\ControleExemplar;
189 use App\Material;
190 use App\Pg;
191 
192 class TesteCargaPostgres extends Testes {
193     public function __construct($execucaoId, $url, $iteracoes, $registros,
    $algoritmo='algoritmo1', $apagarRegistros = 0) {
194         parent::__construct($execucaoId, $url);
195         $testeArmazenar = new \stdClass();
196         $testeArmazenar->postgres = new \stdClass();
197         $testeArmazenar->postgres->dados = [];
198 
199         if ( $apagarRegistros == 1 ) {
200             ControleExemplar::apagarTudoPg();
201             Material::apagarTudoPg();
202             ControleMaterial::apagarTudoPg();
203             ControleMaterial::reiniciarNumeroControlePg();
```

```
204         }
205
206         $this->iniciar();
207         if ( $algoritmo == 'algoritmo1' ) {
208             $testeArmazenar->postgres->tipo = 'carga1';
209             $this->algoritmo1($iteracoes);
210         }
211         else {
212             $testeArmazenar->postgres->tipo = 'carga2';
213             $this->algoritmo2($iteracoes);
214         }
215         $tempo = $this->finalizar();
216
217         $teste = new \stdClass();
218         $teste->registros = $registros;
219         $teste->tempo = $tempo;
220         $teste->tamanhoBase = Material::obterTamanhoBasePg();
221         $testeArmazenar->postgres->dados[] = $teste;
222         $this->armazenarTeste($testeArmazenar);
223       }
224
225     private function obterSQLControle($numerocontrole) {
226         $data = date('d/m/Y');
227
228         return "INSERT INTO gtccontrolematerial VALUES
    ({$numerocontrole},NULL,'{$data}','{$data}',21,14);";
229     }
230
231     private function obterSQLMaterial($numerocontrole, $registros, $tipo = 1) {
232         $execucaoId = $this->execucaoId;
233         $sqls = [];
234
235         if ( $tipo == 1) {
236             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'041','a','0',NULL,NULL,'Português','PORTUGUES');";
237             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'090','a','0',NULL,NULL,'681.3.07','681.3.07');";
238             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'090','b','0',NULL,NULL,'D232i','D232I');";
239             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'100','a','0',NULL,NULL,'Date, C. J.','DATE, C. J.');";
240             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'245','a','0',NULL,NULL,'Introdução a sistemas de bancos de
    dados','INTRODUCAO A SISTEMAS DE BANCO DE DADOS');";
241             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'260','a','0',NULL,NULL,'Rio de Janeiro','RIO DE JANEIRO');";
242             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'260','b','0',NULL,NULL,'Elsevier','ELSEVIER');";
243             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'300','a','0',NULL,NULL,'865 p.','865 P.');";
244             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'650','a','0',NULL,NULL,'Informática','INFORMATICA');";
245             $sqls[] = "INSERT INTO gtcmaterial VALUES
    ({$numerocontrole},'650','a','1',NULL,NULL,'Processamento de
    dados','PROCESSAMENTO DE DADOS');";
246         }
247         elseif ( $tipo == 2) {
248             // Omitido.
249         }
250         elseif ( $tipo == 3 ) {
251             // Omitido.
252         }
253         elseif ( $tipo == 4 ) {
254             // Omitido.
255         }
256         elseif ( $tipo == 5 ) {
257             // Omitido.
258         }
```

```
259         return $sqls;
260     }
261
262     private function obterSQLExemplar($numeroControle) {
263         $data = date('d/m/Y');
264         $sqls = [];
265         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '1', 21, 14, '{$data}', 0);";
266         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '2', 21, 14, '{$data}', 0);";
267         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '3', 21, 14, '{$data}', 0);";
268         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '4', 21, 14, '{$data}', 0);";
269         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '5', 21, 14, '{$data}', 0);";
270         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '6', 21, 14, '{$data}', 0);";
271         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '7', 21, 14, '{$data}', 0);";
272         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '8', 21, 14, '{$data}', 0);";
273         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '9', 21, 14, '{$data}', 0);";
274         $sqls[] = "INSERT INTO gtccontroleexemplar VALUES ({$numeroControle},
    '10', 21, 14, '{$data}', 0);";
275         return $sqls;
276     }
277
278        public function algoritmo1($registros) {
279         $db = Pg::getInstance();
280         $tipo = 1;
281
282               for ( $i = 0; $i < $registros; $i++ ) {
283                      $numeroControle =
    ControleMaterial::proximoNumeroDeControlePg();
284             $sqlControle = $this->obterSQLControle($numeroControle);
285             $db->execute($sqlControle);
286             $sqlsMaterial = $this->obterSQLMaterial($numeroControle, $registros,
    $tipo);
287
288             if ( $tipo == 5 ) {
289                 $tipo = 1;
290             }
291             else {
292                 $tipo++;
293             }
294             foreach ( $sqlsMaterial as $sql ) {
295                 $db->execute($sql);
296             }
297             $sqlsExemplar = $this->obterSQLExemplar($numeroControle);
298
299             foreach ( $sqlsExemplar as $sql ){
300                 $db->execute($sql);
301             }
302               }
303        }
304
305     public function algoritmo2($registros) {
306         $db = Pg::getInstance();
307         $tipo = 1;
308         $iteracoes = [];
309
310         if ( $registros > 10000 ) {
311             $registrosQuantidade = $registros;
312             $quantidade =  ceil($registrosQuantidade / 10000);
313             for ( $i=0; $i < $quantidade; $i++ ) {
314                 if ( $registrosQuantidade > 10000 ) {
```

```
315                     $iteracoes[$i] = 10000;
316                     $registrosQuantidade -= 10000;
317                 }
318                 else {
319                     $iteracoes[$i] = $registrosQuantidade;
320                 }
321             }
322         }
323         else {
324             $iteracoes[] = $registros;
325         }
326 
327         foreach ( $iteracoes as $iteracao ) {
328             $sqls = [];
329             for ( $i = 0; $i < $iteracao; $i++ ) {
330                 $numeroControle = ControleMaterial::proximoNumeroDeControlePg();
331                 $sqls[] = $this->obterSQLControle($numeroControle);
332                 $material = $this->obterSQLMaterial($numeroControle, $registros,
    $tipo);
333                 $sqls[] = $material[0];
334                 $sqls[] = $material[1];
335                 // Omitido.
336                 $sqls[] = $material[9];
337 
338                 if ( $tipo == 5 ) {
339                     $tipo = 1;
340                 }
341                 else {
342                     $tipo++;
343                 }
344                 $exemplar = $this->obterSQLExemplar($numeroControle);
345                 $sqls[] = $exemplar[0];
346                 $sqls[] = $exemplar[1];
347                 // Omitido.
348                 $sqls[] = $exemplar[9];
349             }
350             $sql = implode("\n", $sqls);
351             $db->execute($sql);
352         }
353     }
354 }
355 
356 ?>
357 
358 <?php
359 namespace App\Testes;
360 
361 include_once('Testes.php');
362 include_once('../Material.php');
363 include_once('../MongoDB.php');
364 include_once('../../config/db.php');
365 
366 use App\Material;
367 use App\MongoDB;
368 
369 class TesteVolumeMongo extends Testes {
370     public function __construct($execucaoId, $url, $registros) {
371         parent::__construct($execucaoId, $url);
372 
373         $testeArmazenar = new \stdClass();
374         $testeArmazenar->mongo = new \stdClass();
375         $testeArmazenar->mongo->tipo = 'volume';
376         $testeArmazenar->mongo->dados = [];
377 
378         $this->iniciar();
379         $this->algoritmo1();
380         $tempo = $this->finalizar();
381 
```

```
382         $teste = new \stdClass();
383         $teste->registros = $registros;
384         $teste->tempo = $tempo;
385         $testeArmazenar->mongo->dados[] = $teste;
386 
387         $this->armazenarTeste($testeArmazenar);
388     }
389 
390     public function algoritmo1() {
391         $db = MongoDB::getInstance();
392         $db->find('material', []);
393     }
394 }
395 
396 ?>
397 
398 <?php
399 
400 namespace App\Testes;
401 
402 include_once('Testes.php');
403 include_once('../ControleMaterial.php');
404 include_once('../Material.php');
405 include_once('../Pg.php');
406 include_once('../../config/db.php');
407 
408 use App\ControleMaterial;
409 use App\Material;
410 use App\Pg;
411 
412 class TesteVolumePostgres extends Testes {
413      public function __construct($execucaoId, $url, $registros) {
414          parent::__construct($execucaoId, $url);
415 
416         $testeArmazenar = new \stdClass();
417         $testeArmazenar->postgres = new \stdClass();
418         $testeArmazenar->postgres->tipo = 'volume';
419         $testeArmazenar->postgres->dados = [];
420 
421         $this->iniciar();
422         $this->algoritmo1();
423         $tempo = $this->finalizar();
424 
425         $teste = new \stdClass();
426         $teste->registros = $registros;
427         $teste->tempo = $tempo;
428         $testeArmazenar->postgres->dados[] = $teste;
429 
430         $this->armazenarTeste($testeArmazenar);
431     }
432 
433     private function obterSQLControle() {
434         return "SELECT * FROM gtccontrolematerial;";
435     }
436 
437     private function obterSQLMaterial() {
438         return "SELECT * FROM gtcmaterial;";
439     }
440 
441     private function obterSQLExemplar() {
442         return "SELECT * FROM gtccontroleexemplar;";
443     }
444 
445     public function algoritmo1() {
446         $db = Pg::getInstance();
447 
448        $sqlMaterialControle = $this->obterSQLControle();
449         $sqlMaterial = $this->obterSQLMaterial();
```

```
450         $sqlExemplar = $this->obterSQLExemplar();
451
452         $db->execute($sqlMaterialControle);
453         $db->execute($sqlMaterial);
454         $db->execute($sqlExemplar);
455     }
456 }
457
458 ?>
459
460 <?php
461 namespace App\Testes;
462
463 include_once('TesteVolumePostgres.php');
464
465 class TesteStressPostgres extends TesteVolumePostgres {
466     public function __construct() {
467     }
468 }
469
470 $executar = $argv[1];
471 $execucaoid = $argv[2];
472 $url = $argv[3];
473 $registros = $argv[4];
474 $acessos = $argv[5];
475 $quantidadeAtualRegistros = $argv[6];
476
477 if ( $executar == '1' ) {
478     $testeObjeto = new Testes($execucaoid, $url);
479     $testeObjeto->iniciar();
480
481     for ($i = 1; $i <= $acessos; ++$i) {
482         $pid = pcntl_fork();
483
484         if (!$pid) {
485             $testeStress = new TesteStressPostgres();
486             $testeStress->algoritmo1();
487             exit($i);
488         }
489     }
490
491     while (pcntl_waitpid(0, $status) != -1) {
492         $status = pcntl_wexitstatus($status);
493     }
494
495     $tempo = $testeObjeto->finalizar();
496
497     $testeArmazenar = new \stdClass();
498     $testeArmazenar->postgres = new \stdClass();
499     $testeArmazenar->postgres->tipo = 'stress';
500     $testeArmazenar->postgres->dados = [];
501
502     $teste = new \stdClass();
503     $teste->registros = $quantidadeAtualRegistros;
504     $teste->tempo = $tempo;
505     $teste->acessos = $acessos;
506     $testeArmazenar->postgres->dados[] = $teste;
507     $testeObjeto->armazenarTeste($testeArmazenar);
508 }
509 ?>
510
511 <?php
512 namespace App\Testes;
513
514 include_once('TesteVolumeMongo.php');
515
516 class TesteStressMongo extends TesteVolumeMongo {
517     public function __construct() {
```

```
518     }
519 }
520
521 $executar = $argv[1];
522 $execucaoid = $argv[2];
523 $url = $argv[3];
524 $registros = $argv[4];
525 $acessos = $argv[5];
526 $quantidadeAtualRegistros = $argv[6];
527
528 if ( $executar == '1' ) {
529     $testeObjeto = new Testes($execucaoid, $url);
530     $testeObjeto->iniciar();
531
532     for ($i = 1; $i <= $acessos; ++$i) {
533         $pid = pcntl_fork();
534
535         if (!$pid) {
536             $testeStress = new TesteStressMongo();
537             $testeStress->algoritmo1();
538             exit($i);
539         }
540     }
541
542     while (pcntl_waitpid(0, $status) != -1) {
543         $status = pcntl_wexitstatus($status);
544     }
545
546     $tempo = $testeObjeto->finalizar();
547
548     $testeArmazenar = new \stdClass();
549     $testeArmazenar->mongo = new \stdClass();
550     $testeArmazenar->mongo->tipo = 'stress';
551     $testeArmazenar->mongo->dados = [];
552
553     $teste = new \stdClass();
554     $teste->registros = $quantidadeAtualRegistros;
555     $teste->tempo = $tempo;
556     $teste->acessos = $acessos;
557     $testeArmazenar->mongo->dados[] = $teste;
558     $testeObjeto->armazenarTeste($testeArmazenar);
559 }
560 ?>
561
562 <?php
563 namespace App\Testes;
564
565 class Testes {
566        protected $inicial;
567     protected $final;
568     protected $tempo = 0;
569     protected $url = '';
570     protected $execucaoId = 0;
571
572     public function __construct($execucaoId, $url) {
573               $this->execucaoId = $execucaoId;
574         $this->url = $url;
575        }
576
577     public function armazenarTeste($teste) {
578         $dados = json_encode($teste);
579         $execucaoId = $this->execucaoId;
580         $url = $this->url . "execucaoid={$execucaoId}&dados={$dados}";
581         file_get_contents($url);
582     }
583
584     public function iniciar() {
585         $this->inicial = $this->obterMicrotime();
```

```
586     }
587
588     public function finalizar() {
589         $this->final = $this->obterMicrotime();
590         $this->tempo =  number_format(($this->final - $this->inicial), 6);
591         return $this->tempo;
592     }
593
594     protected function obterMicrotime() {
595         $microtime = explode(" ", microtime());
596         $time = $microtime[0] + $microtime[1];
597         return $time;
598    }
599 }
600 ?>
601
602 <?php
603 namespace App\Testes;
604
605 $execucaoid = $argv[1];
606 $url = $argv[2];
607 $iteracoes = $argv[3];
608 $registros = $argv[4];
609 $carga1 = $argv[5];
610 $carga2 = $argv[6];
611 $apagar = $argv[7];
612 $volume = $argv[8];
613 $stress = $argv[9];
614 $acessos = $argv[10];
615 $quantidadeAtualRegistros = $argv[11];
616
617 include_once('TesteCargaPostgres.php');
618 include_once('TesteCargaMongo.php');
619 include_once('TesteVolumePostgres.php');
620 include_once('TesteVolumeMongo.php');
621
622 system('touch .executando');
623
624 if ( $carga1 == 1) {
625        $testeCarga = new TesteCargaPostgres($execucaoid,  $url, $iteracoes,
    $registros, 'algoritmo1', $apagar);
626        $testeCarga = new TesteCargaMongo($execucaoid,  $url, $iteracoes,
    $registros, 'algoritmo1', $apagar);
627 }
628
629 if ( $carga2 == 1) {
630        $testeCarga = new TesteCargaPostgres($execucaoid,  $url, $iteracoes,
    $registros, 'algoritmo2', $apagar);
631        $testeCarga = new TesteCargaMongo($execucaoid,  $url, $iteracoes,
    $registros, 'algoritmo2', $apagar);
632 }
633
634 if ( $volume == 1 ) {
635        $testeVolume = new TesteVolumePostgres($execucaoid, $url, $registros);
636        $testeVolume = new TesteVolumeMongo($execucaoid, $url, $registros);
637 }
638
639 if ( $stress == 1) {
640        system("php TesteStressPostgres.php {$stress} {$execucaoid} {$url}
    {$registros} {$acessos} {$quantidadeAtualRegistros}");
641        system("php TesteStressMongo.php {$stress} {$execucaoid} {$url}
    {$registros} {$acessos} {$quantidadeAtualRegistros}");
642 }
643
644 system('rm .executando');
645
646 ?>
```